

AD-A238 049



RL-TR-91-76, Vol IV (of four)
Final Technical Report
June 1991

DTIC
ELECTF
JUL 10 1991
S C D



2

EXPERT SYSTEMS ON MULTIPROCESSOR ARCHITECTURES Technical Reports

Stanford University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. 5291

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

91-04338



The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

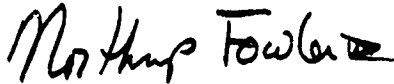
Rome Laboratory
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

91 7 8 023

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-91-76, Volume IV (of four) has been reviewed and is approved for publication.

APPROVED:



NORTHROP FOWLER III
Project Engineer

APPROVED:



RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER:



RONALD RAPOSO
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL(COE) Griffiss AFB, NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

EXPERT SYSTEMS ON MULTIPROCESSOR ARCHITECTURES,
Technical Reports

Edward A. Feigenbaum
Robert Engelmores
H. Penny Nil
James P. Rice

Contractor: Stanford University
Contract Number: F30602-85-C-0012
Effective Date of Contract: 14 March 1985
Contract Expiration Date: 31 March 1990
Short Title of Work: Concurrent Expert Systems
Architecture
Program Code Number: OE20
Period of Work Covered: Mar 85 - Mar 90

Principal Investigator: Edward A. Feigenbaum
Phone: (415) 723-4878

RL Project Engineer: Northrup Fowler III
Phone: (315) 330-7794

Approved for public release; distribution unlimited.

This research was supported by the Defense Advanced
Research Projects Agency of the Department of
Defense and was monitored by Northrup Fowler III,
RL (COE), Griffiss AFB NY 13441-5700 under Contract
F30602-85-C-0012.



Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE June 1991		3. REPORT TYPE AND DATES COVERED Final Mar 85 - Oct 90	
4. TITLE AND SUBTITLE EXPERT SYSTEMS ON MULTIPROCESSOR ARCHITECTURES, Technical Reports				5. FUNDING NUMBERS C - F30602-85-C-0012 PE - 62301E PR - E291 TA - 00 WU - 01	
5. AUTHOR(S) Edward A. Feigenbaum, Robert Engelmores, H. Penny Nii and James P. Rice					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Knowledge Systems Laboratory Stanford University 701 Welch Rd, Bldg C Palo Alto CA 94304				8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency 1400 Wilson Blvd Arlington VA 22209-2308 Rome Laboratory (COE) Griffiss AFB NY 13441-5700				10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-91-76, Vol IV (of four)	
11. SUPPLEMENTARY NOTES Rome Laboratory Project Engineer: Northrup Fowler III/COE/(315) 330-7794					
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) This final report documents the results of a five-year investigation of methods for achieving higher performance for knowledge-based systems through the design of innovative software and hardware systems architectures. Volume I summarizes the work performed and lessons learned, and serves as an annotated index to the set of over 50 project technical reports. Volumes II through IV contain the project technical reports. NOTE: Rome Laboratory/RL (formerly Rome Air Development Center/RADC)					
14. SUBJECT TERMS Multiprocessor Architectures, Artificial Intelligence, Blackboard Systems				15. NUMBER OF PAGES 558 16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL		

Table of Contents for Volume 4

[Rice 86a]	Poligon, A System for Parallel Problem Solving.	4-1
[Rice 86b]	The Poligon User's Manual.	4-19
[Rice 88a]	Problems with Problem-Solving in Parallel: The Poligon System.	4-139
[Rice 88b]	The Elint Application on Poligon: The Architecture and Performance of a Concurrent Blackboard System.	4-165
[Rice 88c]	The Advanced Architectures Project.	4-176
[Rice 89a]	See How They Run... The Architecture and Performance of Two Concurrent Blackboard Systems.	4-198
[Rice 89b]	The Design of a High Performance, Concurrent Problem Solving System...and many Lessons Learned on the Way.	4-219
[Saraiya 86]	AIDE: A Distributed Environment for Design and Simulation.	4-297
[Saraiya 88]	A Shared Memory Lisp Package for CARE.	4-324
[Saraiya 89]	Design and Performance Evaluation of a Parallel Report Integration System.	4-337
[Saraiya 90a]	SIMPLE/CARE. An Instrumented Simulator for Multiprocessor Architectures.	4-360
[Saraiya 90b]	The LAMINA Programming Model: A Worked Example.	4-394
[Schoen 86]	The CAOS System.	4-433
[Thapar 89a]	Design and Implementation of a Distributed Directory Cache Coherence Protocol.	4-502
[Thapar 89b]	Distributed Cache Coherence for Large Scale Shared Memory Multiprocessors.	4-527
[Thapar 90]	Cache Coherence for Large Scale Shared Memory Multiprocessors	4-542

Poligon, A System for Parallel Problem Solving

by
James Rice

(Rice@Sumex-Aim.Stanford.Edu)

**Knowledge Systems Laboratory
Stanford University
701 Welch Road
Palo Alto, CA 94304**

The author gratefully acknowledges the support of the following funding agencies for this project; DARPA/RADC, under contract F30602-85-C-0012; NASA, under contract number NCC 2-220; Boeing Computer Services, under contract number W-266875.

Summary

The **Poligon** system is a new, domain-independent language and attendant support environment, which has been designed specifically for the implementation of applications using a Blackboard-like problem-solving framework in a parallel computational environment.

This paper describes the Poligon system and the Poligon language, its salient and novel features. Poligon is compared with other approaches to the programming of parallel systems.

1. Introduction

The larger project of which Poligon is only a small part will not be discussed here in any detail. Design decisions made in other parts of the project will be held to be axiomatic, though some mention of these decisions will be made in order to show the motivation for the features of Poligon. The primary objective of the overall project is to achieve significant speedup of knowledge based systems, particularly those directed at real-time signal understanding.

The purpose of the Poligon language is to express the problem solving behaviour of human experts in order to map them onto a problem solving framework, which will run on simulated parallel hardware.

The fields of knowledge representation and problem solving are rich and complex. This paper will not go into any great detail in describing the problem solving processes involved. Poligon tries usefully to express knowledge both in a declarative and procedural sense, through rules [Davis 77]; and in a structural sense, through the configuration of the solution space. These will be described below.

Some crucial design criteria and early design commitments have affected the development of Poligon, the consequences of which will be described in this paper. These can be summarised as follows.

- Poligon is intended to be a language for both problem solving and the general purpose programming necessary to support it. Unlike most programs, Poligon programs must also address the problems of real-time processing, including asynchronous events and input data backup. Poligon, therefore, must assist in this respect.
- The overall project's strategy is to solve problems significantly faster than existing systems through the exploitation of parallelism. Poligon is targeted at a MIMD, distributed-memory, message-passing machine with ~thousands of processors. This hardware gives direct support for futures, remote objects and such efficient message-passing strategies as *Broadcast* and *Multicast* so as to take full advantage of its processor interconnection network.
- A consequence of the desire to achieve a significant order of parallelism in Poligon programs is that many of the control mechanisms used in serial problem solving systems, such as schedulers and event queues, have been discarded because they are highly serial. Most actions in Poligon programs are, therefore, performed asynchronously. Rules, the primary mechanism in Poligon for describing things

and for getting things done, are activated as daemons. Much of the work in Polygon is aimed at providing mechanisms to cope with this chaotic behaviour.

This paper contains the following:

- A discussion of related work in parallel languages.
- A discussion of the design approach guiding the development of Polygon.
- A description of the abstraction mechanisms provided by the Polygon system with some small examples.
- Some concluding remarks.
- References for further reading on the subject.

1.1. Knowledge Representation and Problem Solving in Polygon

The primary purpose of this paper is to discuss the Polygon language. It is, however, not possible completely to divorce this from the underlying hardware and from its purpose; knowledge representation and problem solving.

Polygon can be described loosely as a "Blackboard System". What this means in practice is that the problem solving metaphor of Polygon is one of cooperating experts gathered around a blackboard, posting ideas about their deductions on the blackboard. For an exposition on the term "Blackboard System" the reader is encouraged to read [Nii 86]. Polygon tries usefully to express knowledge both in a declarative and procedural sense, through rules and functions; and in a structural sense, through the configuration of the solution space on the blackboard. In particular, the term "blackboard" will be used to describe the set of all of the nodes in the solution space of the system.

The suggestion that Polygon is a blackboard system is a little controversial. There are a number of respects in which this is not a satisfactory label. This term will, however, be used freely from now on for lack of a better label. The reader is encouraged to substitute for the term "Blackboard system" any term, such as "Frame System" which seems best to fit his mental model of what is being described.

1.2. Polygon's Model of Parallelism

It seems appropriate here to describe Polygon's model of parallelism. In its simplest form this can be thought of as *An Element in the Solution Space as a Processor*.

This gives some idea of the granularity that is being sought. It is, however, by no means the most efficient way to implement Polygon. Polygon programs want to be able to execute rules and parts of rules associated with a particular *Node* in the solution space in parallel. These rule activations need processors, on which to execute.

Thus a modified version of Polygon's model of parallelism could be *A Rule Activation as a Process, with sufficient processors to cope with the parallelism exhibited by the rule during its activation*. This tends towards a mapping of solution space elements onto a cluster of processors to service the rule activations. In practice, however, a number of nodes might be folded over the same set of processors, either because nodes become quiescent or because the load balancing in the system is sub-optimal.

2. Related Work

Work in this field falls into two distinct categories; work on parallel knowledge based systems and work on languages for parallel symbolic computation. The former is, at present, a very sparse field and, will not be discussed here, though some references are given in Section 6. The latter is much more highly developed.

Much work is already being done on parallel languages for general computation. Amongst these languages are **Actors**, **MultiLisp** and **QLisp** on the one hand and concurrent logic programming languages and purely functional languages on the other. Often missing from this work is a thrust toward the investigation of large applications in parallel domains, for instance the development of parallel knowledge representation and problem solving systems. This is, of course, what Poligon attempts to do. This section will discuss briefly Actors, QLisp and Multilisp, since these are the parallel symbolic computation languages which are most relevant to the development of Poligon and the software which lies beneath it.

2.1. Actors

Actors [Hewitt 73] probably come the closest in their behaviour to Poligon, at least at an implementation level. Actors are independent, asynchronously communicating objects. As is the way with purely object oriented systems they communicate only through message passing and have tightly defined operations. The mutual control of Actors and parallelism is achieved by the support of procedure call and coroutine model message passing. The modularity afforded by this sort of programming metaphor may well be especially useful for the programming of distributed-memory, message-passing hardware, since having a close match between the hardware and software metaphors is likely to achieve better performance. It is not in any way surprising that the operating system level software, which underlies Poligon, is founded on many of the same principles as Actors. It has yet to be seen whether this programming methodology is able in practice to extract significant amount of parallelism from problems, though clearly this project hopes that it is.

2.2. MultiLisp and QLisp

MultiLisp [Halstead 84] and **QLisp** [Gabriel 84] are lumped together because, at least in some senses, they have strong generic resemblances. They are both, at the user level, extensions to existing Lisp dialects which provide mechanisms for the expression of parallelism, such as parallel Let constructs and parallel function argument evaluation (QLet and PCall). It is assumed by both of these systems that the hardware at which they are targeted is a form of shared-memory multiprocessor. Although there is no particular reason why such systems could not be implemented on a distributed-memory system, they are optimised for shared-memory multiprocessors. These are currently the most readily available form of multiprocessor. They would, however, need significant extensions in order to be able to exploit a distributed-memory system as is shown in **CAREL** [Davies 86], an implementation of QLisp for distributed-memory machines. The assumption of shared-memory, MIMD processors in these systems imposes constraints on the languages. They assume, at least to an extent, that processes will be expensive and that the user must have control over their creation. Poligon assumes quite the opposite.

3. The Design of Poligon

Poligon will be discussed first in terms of the way in which the language relates to the problems being solved and its underlying systems. Next the language will be discussed in terms of the requirements for languages in general and parallel languages in particular.

3.1. Background and Motivation

The philosophy behind the design of Poligon comes from intellectual and pragmatic pressures. It attempts to steer a middle course between the extreme purism of applicativists and the extreme pragmatism of the proponents of side-effects.

From the outset, the project was oriented towards real-time problem solving. Blackboard systems are well known to be of interest as tools in the knowledge engineer's toolkit. Little work has been done to investigate the appropriateness of the blackboard metaphor to parallel execution or the meaning of parallel blackboard systems, though it is frequently claimed that they are full of latent parallelism. The excellent formal properties of pure applicative and logic languages may well be of little use in a system which, for whatever reasons, needs to express side-effects and which has to cope with real-time constraints. Poligon is a system in which some of the formal rigour of truly applicative systems has been put aside in favour of a pragmatic approach to the exploitation of parallelism.

The **BB1** project [Hayes-Roth 85], also a project at the HPP, is an attempt to investigate the behaviour of highly controlled problem solving systems. It attempts to use a great deal of meta-knowledge and makes significant use of globality of reference in order to support an holistic view of its solution space, thus providing a basis for meta-level reasoning. The Poligon project is an attempt to investigate quite the reverse. Poligon has very little support for meta-knowledge and allows no global data or global view of the solution space whatsoever. The purpose of this experiment is to determine whether a system, unconstrained by a great deal of serialising control knowledge, might still be able to find useful answers faster than an highly controlled system, such as BB1, which would be extremely difficult to speed up significantly through parallelism.

The Poligon system pictures the elements in its solution space as processes resident on processors distributed across a grid, with the code necessary for them intimately associated with them. Because no global control is permitted in Poligon the activation of rules is necessarily completely daemon-driven.

The project hopes to achieve significant speed-up through parallelism. This can be done only if much parallelism is extracted from the problem. Ideally, the system would try to achieve its parallelism by exploiting parallelism in the program's implementation at a very fine grain. This can, in principle, extract the maximum amount of parallelism available. On its own it has drawbacks, however. The costs of processes and the problems of synchronisation at a fine grain size make it difficult to exploit such parallelism without the use of hardware mechanisms significantly different from those available with prevailing technologies. This approach is also only part of the story. It neglects the fact that a properly parallel decomposition of the source problem is crucial to finding a lot of parallelism. One could summarise the problems, therefore, as expressing the problem in a sufficiently parallel fashion and the matching of the parallelism in the program to the grain size of the underlying hardware. Poligon addresses these issues.

Parallelism is very hard to find in conventional programs. Applicative systems have an advantage in this respect because of their relative lack of need to express parallelism explic-

itly. Their unchanging semantics when parallelism is introduced eases matters considerably. Polygon has attempted to learn from this and has pure applicative semantics in a number of areas but takes a different approach to the finding of parallelism in programs. It attempts to execute everything in parallel that it can and leaves it to the programmer to find any serial dependencies.

When the parallelism in a program is user-defined, problems can result from an inappropriate match between the granularity of the parallelism expressed in the program and the granularity of the underlying machine. In systems of the size and complexity of a typical Polygon application such a match would be particularly difficult to find because of the large number of processors involved and because it would be difficult for the user to keep track of the location of his data in the processor array. These characteristics are a consequence of the highly variable and data dependent state of the solution space in such programs. Polygon, because of its structure, should be able largely to obviate such granularity mismatches because parallelism is defined and controlled by the system and the Polygon system is closely matched to the granularity of the underlying system.

It is often thought that problems suitable for solution by means of the blackboard model tend to partition their solution spaces into what look rather like pipe-lines. Pipe-lines are, of course a well known form of parallelism. In practice pipes in such systems are not pipes in the normal sense, since they are more like "leaky" pipes. It is one of the prime objectives of these systems to reduce the amount of data as it percolates up through the abstraction hierarchy of the solution space. Because of the reduction in the data rate flowing in these pipes the contention problems that one might expect when pipes are connected into trees, as they often are, are alleviated.

A significant limitation of the performance of pipelines is that, at best, the parallelism that they can produce is proportional to the length of the pipe. This would typically be only of the order of half a dozen sections. This is clearly not the "orders of magnitude" of performance improvement that we all hope for. In practice, though, given a large enough problem, it is often possible to set up a large number of these pipes side-by-side. It is one of the major objectives of the Polygon language to encourage, facilitate and reward the decomposition of problems so that this form of independence can be exploited, so that such pipes will be created by the system.

3.2. Language Requirements

Polygon is a language which is by no means directed at general computation. It is nevertheless intended to be used for the solution of large, complex problems on distributed-memory parallel hardware. The following is a brief list of the ways in which Polygon attempts to address some of the primary requirements of programming languages.

- The language should provide a tangible method of expressing the ideas of the programmer.

The Polygon language has been written with considerable input from those with experience in problem solving systems in the application domains at which it is targeted. It is therefore intended to match the ideas of the "Expert", whose knowledge is to be encoded, but in a domain independent way.

- The compiler¹ should provide a mapping between the language and the underlying systems, be they hardware or software.

Poligon's compiler compiles Poligon language source into code understood by the underlying *Lisp* system and the concurrent object-oriented operating system running on its target hardware.

- The language should abstract the programmer from its underlying systems.

The Poligon system shields the user from all aspects of the underlying hardware such as the topology of the processor network, the message-passing behaviour of the hardware and the location of any code or data within the network.

- The language should provide mechanisms for the exploitation of the underlying systems to good effect.

The underlying hardware and software systems are exploited in a number of ways in Poligon. Firstly the language encourages the user naturally to decompose his problem into a form which will map efficiently onto the underlying hardware. Secondly the language offers a number of application-independent, high-level constructs, which are designed to exploit the hardware to the full. These topics are covered more fully in Section 4.

- The language should allow the development of software faster than would be the case if it were to be developed in a less abstract form.

Considerable effort has been spent on making the Poligon language a high level way to describe the solutions to parallel knowledge based system problems. A high level language with such features as infix, user-definable operators and user definable syntax, provides a natural way for the expert to implement his knowledge.

Much effort has been spent also on integrating the Poligon system cleanly into the program support environment of the Lisp Machines on which it runs. For instance, incremental compilation is supported from within the editor.

- The language should assist the development of reliable, maintainable and modular software.

Language features are provided to minimise the possibility of inconsistent modifications to the source code and the structure of the language and its semantics are defined in a manner which minimises the probability of complex bugs being introduced by asynchronous side-effects.

A sophisticated set of debugging facilities is provided. A system that emulates the semantics of full, parallel Poligon programs as closely as possible in a serial environment has been produced. The user is able to debug his program serially to remove all possible serial bugs and bugs due to the non-deterministic execution order of Poligon programs before it is ported to the full parallel environment.

¹The term *Compiler* is used in its most general sense here, perhaps an interpreter or a machine which is clever enough to execute the language specified directly.

In addition to these requirements a language targeted at parallel hardware should have a number of attributes which reflect the parallel nature of the target hardware.

- The language should address the granularity of the hardware.

Poligon is closely matched to the granularity of the hardware at which it is targeted. It is generally expected that the solution space of the problems addressed by Poligon programs will have of the order of thousands of nodes. This is of the same order as the granularity of the hardware.

- The language should provide a mechanism for the extraction of parallelism from programs and from the programmer.

Poligon extracts parallelism from programs and the programmer in two main ways. First the decomposition of the problem is encouraged to be as modular as possible. Secondly the semantics of Poligon programs are such that almost all of the program can be executed in parallel without changing their behaviour from that seen during serial execution. This allows the system to execute most operations in parallel if it has the resources to do so.

- The language should, where appropriate, shield the programmer from those details of the hardware which are particular to parallel computing engines, such as topology.

The hardware, on which Poligon programs runs, causes Poligon programs to have to cope with communication between solution space elements on different processor sites. All such message passing is hidden from the user. In fact the Poligon language has no concept of message-passing at all.

Futures are used for all remote operations in the user's program. The hardware implements these such that there is no efficiency penalty associated with creating futures for such remote accesses. The Poligon language copes with these invisibly to the programmer.

As can be seen quite easily from the above one of the factors that must be well understood before a language is designed is the general purpose of the language and the level of generality that is expected of programs written in it. A language, whose sole purpose is the expression of solutions to huge matrix problems on systolic hardware might well be justified in expecting the programmer to express, at quite a low level, the mapping of the program onto the hardware provided. This is less likely to be a reasonable expectation of a language targeted at the solution of large, complex problems of an unpredictable, dynamically-varying or data-dependent nature. Poligon is a fairly general purpose programming language with a very definite bias.

4. Abstractions in Poligon

To cope with Poligon's view of parallelism and with the chaotic execution of rules (see Section 1) a number of linguistic abstractions are provided. Poligon provides abstractions for knowledge representation, control, data, parallelising, real-time and side-effect control. These will be described briefly in this section.

4.1. Knowledge Representation

Knowledge is traditionally represented in blackboard systems in a number of ways, listed below.

- Declarative Knowledge is encoded in Rules.
- Procedural Knowledge is encoded in procedures.
- Knowledge concerning the sequencing of activities is encoded in the scheduling mechanism.
- Knowledge about the structure of the solution space is encoded by the definition of the structure of the blackboard.
- Knowledge about relationships between the objects in the system is often encoded using a Link mechanism.

These all represent knowledge about the application domain. In addition, there is in any program a large body of implicit knowledge concerning the semantics of assignment, sequencing and the system's function as a whole, especially in for systems with poor formal properties. This will not be discussed here. The Poligon language does, however, go to considerable effort to make the semantics of the Poligon system as clear as possible.

4.1.1. Declarative Knowledge

The encoding of *Declarative Knowledge* in blackboard systems is conventionally done in *Rules*¹, which exist within scheduling units known as *Knowledge Sources*. Poligon also has the concept of Rules and Knowledge Sources, though their meaning is somewhat different. Unlike serial blackboard systems, the rules in a Poligon system are activated autonomously and asynchronously.

Existing blackboard systems usually suffer from a confusion and overloading in the semantics and purpose of knowledge sources. It is useful to collect one's knowledge of one subject together into one chunk. These chunks are knowledge sources. Sadly, the implementors of blackboard system frameworks often think of knowledge sources as scheduling units and thus design their scheduling strategies around the idea of the "invocation of knowledge sources", even though it is by no means necessarily the case that it is appropriate to schedule all of knowledge in a chunk at the same time. This has a detrimental effect on the modularity of the system.

In Poligon, knowledge sources are used as linguistic and software engineering abstractions provided for the programmer in order to allow him to collect related knowledge together. There are no scheduling semantics associated with knowledge sources in Poligon. Because of the underlying system's daemon-like rule triggering mechanism the rule writer is allowed completely to decouple the concept of *scheduling* from the concept of *chunks of knowledge*.

¹The term *Rule* is used here in the sense of "Pattern/Action pairs". It should be noted that these are quite unlike the structures called rules used, for instance, in Prolog. Pattern/Action rules move towards a solution to their problem by performing side-effects on their environment, in this case the blackboard, not through unification.

Rules are activated as a result of "events" happening to the fields of nodes (see Section 4.3.1). These events can be caused either by a write operation to a field, by a semaphore being waved at a field or by the real-time clock.

A powerful *Expectation* mechanism is provided, which allows the dynamic placement and specialisation of rules. An Expectation is a way of expressing model-based knowledge. Given a particular model of the behaviour of a system, certain changes might be expected if the model's interpretation of the world is correct. Expectations allow such changes to be watched and even allow their associated rules to be triggered if the changes do not happen in a given time. Such expectations can be placed to watch for events happening, or not happening, in specific places on the blackboard, at specific times. Expectations provide a focussing mechanism¹ and, coupled with the system's ability to trigger² rules and "time-out" unsatisfied Expectations on the basis of the real-time clock, Polygon allows complex time-critical knowledge to be expressed and applied simply.

An example rule is shown in Figure 1.

The following is a trivial example rule, which shows a small set of the features of Polygon. This rule could be interpreted as saying; "If the most recent two phonemes that have been seen are "oo" and "ph" then the word is "foo". Having concluded this the rule finds the set of sentence components, which represent potential conclusions of the word "foo", and sets them so that they are no longer marked as hypothetical. It also makes a Sentence-Component type node, which represents the word "foo", which has been found.

```
Rule : Find-the-word-Foo
  Class : Phoneme
    { Class of nodes with which the rule will be associated }
  Field : uncorrelated-phonemes
    { Try to activate this rule when this field is changed }
  Definitions :
    all-phonemes-in-order ≡ The-Phoneme≠uncorrelated-phonemes
      { The operator "⊕↑" returns all values in a field in }
      { time order. The-Phoneme represents the node, that }
      { triggered this rule }
    most-recent-phoneme ≡ all-phonemes-in-order.Head
    next-most-recent-phoneme ≡ all-phonemes-in-order.Tail.Head
      { Head and Tail are like CAR and CDR only they operate }
      { on lists, Lazy lists and Bags }
  Condition Part :
    When : all-phonemes-in-order.length-of-list ≥ 2
      { The "When" part is a locally evaluable precondition }
    If : most-recent-phoneme.Sound = "oo"
      And next-most-recent-phoneme.Sound = "ph"
      { The precondition for the Rule }
  Action Part :
    Definitions :
      new-sentence-component
        ≡ New Instance of Sentence-Component
```

¹It should be noted that the term *Focussing mechanism* is used in a more general sense than by many blackboard systems. There can be any number of such foci all acting in parallel in a Polygon program. The expectation mechanism is another way of applying knowledge in order to take advantage of some local circumstances in order to solve a problem more efficiently or cleanly.

²A rule is said to have been *Triggered* when it is activated so that it tries to evaluate its preconditions and body.

```

    { The creation of the new Sentence-Component node }
    hypothetical-foos ≡
    { A Bag of words, which are "foo" }
      Subset of Words which satisfies
        λ(a-word)
          a-word.hypothesised And a-word.letters = [ f o o ]
        Endλ
    { Process all elements in the Bag hypothetical-foos }
  Changes :
    In Parallel for each a-word in hypothetical-foos
      Change Type      : Update
      Updated Node     : a-word
      Updated Fields   : hypothesised ← nil
    { Set fields of new sentence component in }
    { parallel with updating the elements in the Bag }
  Changes :
    Change Type      : Update
    Updated Node     : new-sentence-component
    Updated Fields   : letters ← [ f o o ]
                      constituents ←
                        List(next-most-recent-phoneme,
                           most-recent-phoneme)

```

All of the actions taken by this rule are performed in parallel, since they are independent of one another, though there is, of course, a serial dependency between the condition part and the action part of the rule.

Fig. 1. An example Poligon rule

4.1.2. Procedural Knowledge

Procedural Knowledge is an all encompassing term usually used indiscriminately to describe both knowledge about the relationships between values (*Functions*) and the mechanisms for performing side-effects and for sequencing events (*Procedures*). This is often a result of such systems being built on top of Lisp systems, which fail to draw distinctions between procedures with side-effects and those without. Poligon does not allow the encoding of arbitrary knowledge into procedures. Only side-effect free functions are allowed. Side-effects are permitted only in the bodies of rules, where they can be controlled.

4.1.3. The Sequencing of Activities

In most blackboard systems knowledge of the required sequencing of events at a macroscopic level is expressed by the implementation of the system's scheduler. In many cases, such as AGE [Nii 79] this scheduler has fixed characteristics and the application has a fixed interface to it. In others, such as MXA [Rice 84], the user can specify the characteristics of the scheduling of knowledge sources. Poligon provides no such mechanism. Since all rules are activated as daemons, entirely asynchronously, the only analogue of scheduling is the implicit sequencing of the activation of rules due to some rules causing changes that trigger other's rules.

4.1.4. The Structure of the Solution Space

Poligon is unlike most blackboard systems in this respect. Most blackboard systems partition the blackboard into *Levels*, which represent the hierarchy of abstraction in the solution space. Poligon uses a much more general representation which is like that of some *Frame*

systems, providing a "Class" mechanism with user defined classes and metaclasses, and compile-time and run-time inheritance. The functionality of the class mechanism in Poligon is a superset of that of the levels provided by most blackboard systems. The programmer can, of course, represent his solution simply using classes as levels in Poligon if he wishes. Classes are discussed more in Section 4.3.1.

4.1.5. Knowledge about Relationships

Relationships between entities in blackboard systems are often expressed by a form of *Link* mechanism. Sometimes this link is not so much a part of the system as a reflection of the fact that fields in nodes can have as their values other nodes in the system. Other systems have more sophisticated mechanisms that express links explicitly and allow property inheritance along links, e.g. BB1, or the propagation of likelihood, e.g. MXA.

Poligon has a number of system defined relationships; "Is an Instance of", "Is a part of" and "Is a subclass of". The user can define arbitrary relationships between nodes on the blackboard. These links allow property inheritance and are, themselves, represented as nodes and so can have attributes in the same way that any other nodes can. Links are therefore first-class citizens in Poligon and they allow Poligon programs to act like semantic nets.

4.2. Control Abstractions

The flow of control is a rather evanescent concept in a Poligon program. Any rule can be triggered at any time. It is important not to think of the control flow in a Poligon program in the same terms as that of a conventional serial program. There is a well defined flow of control within rules; the action part of a rule is activated after "condition part, upon which it is predicated. Apart from this, however, there is no flow of control in any normal sense. It should be noted also that what little flow of control there is only specifies the strict ordering of activities. The execution of a sequence of actions can be interrupted at any time. The size of the atoms for Poligon's atomic actions is very small.

The triggering of rules is controlled by the user associating rules with particular fields of nodes or classes of nodes on the blackboard. The triggering of rules occurs when a field, which is being watched in such a manner, is updated or is semaphored. A semaphore mechanism is provided to allow rules to be triggered without a field being updated. This provides a form of explicit event-based programming, if it is needed.

Clearly one of the objectives of the design of the Poligon language is to provide a language in which it is simple to express logically distinct pieces of knowledge, independent of other such pieces of knowledge. The decomposition of the problem in this manner causes the system to appear to iterate towards the solution of its problem by small, simple and discrete steps, rather than by complex, giant leaps.

4.3. Data Abstractions

Poligon provides a number of distinct data abstractions. One is characteristic of other blackboard systems, one of pure functional languages and one is rather novel.

- The structure of the blackboard is characterised by being made of *Nodes*, elements in the solution space. These have a user-defined, record-like structure.
- Lazy evaluation is supported.

- Bags are supported as data structures, which parallelism enhancing.

Numerous operations are defined for these data abstractions, particularly a number of generic operations which can be applied to lists, lazy lists and bags, which shield the user from the underlying data structures used by the system or by other segments of his program.

4.3.1. The Structure of the Solution Space

The most obvious data abstraction provided by Poligon is similar to that provided by conventional blackboard systems, that is, the *Node* on the blackboard as an element in the solution space. Such nodes are record-like internally. They have named fields, which can often contain multiple values to be associated with that name. Poligon provides this but also goes beyond it.

Conventional blackboard systems, such as AGE, tend to provide nodes on a blackboard divided into groups, often called "Levels". "Levels" themselves are not represented. Arbitrary use of global data, held in global variables, distinct from the blackboard is also allowed.

Poligon has a much more regular representation for data. The nodes are represented as instances of *Classes*. The Classes themselves are represented as Nodes, which "control" their instances. Knowledge concerned with classes as a whole can be associated with these nodes. Shared, global variables are not allowed in Poligon.

Poligon also provides;

Superclasses Classes that provide characteristics to the instances of classes. These can be thought of as templates for the instances.

Metaclasses Classes that provide characteristics to the classes themselves. These can be thought of as templates for the classes.

Thus the classes are themselves instances of metaclasses, which can be user defined, such that instances of a given class can have any number of superclasses, i.e. component templates, and any number of metaclasses, i.e. component templates for their parent class. It is possible to instantiate classes any number of times, as well as their instances.

Automatic property inheritance allows shared data to be located on locally central nodes, which are immediately visible to the interested parties. This distributes shared data in such a manner as will, hopefully, minimise hot-spotting.

An example class declaration, the specification of a template for a class of nodes, is shown below. The declaration defines a class of nodes called *Words*, each instance of which has two fields (slots) called *Letters* and *Sound*. Class Words :

```
Fields :
  Letters
  Sound
```

Extensions to this sort of syntax allows the definition of superclasses and metaclasses within class declarations. The following example defines the class *Sheep*. Each instance of the class *Sheep* will have the characteristics defined for sheep and for mammals. The

class called *Sheep* (an instance, in fact, of the class *Meta-Sheep*) has the characteristics of *types of animals*.

```
Class Types-of-animals :  
  Fields :  
    Rate-Of-Breeding  
  
Class Mammals :  
  Fields :  
    Colour-of-fur  
    Number-of-legs : 4  
  
Class Sheep :  
  Metaclasses : Types-of-animals  
  Superclasses : Mammals  
  Fields :  
    Thickness-of-wool  
    Flock
```

4.3.2. Lazy Evaluation

Lazy Evaluation is supported in the guise of *Lazy Lists*, *Lazy Function Arguments* and in the form of the lazy association of expressions with names. The following is an example of the lazy association of a name with a value. The name *A-Meaningful-Name* is associated with the value of the call to the function *An-Expensive-Function*¹.

```
Definitions :  
  A-Meaningful-Name ≡  
    An-Expensive-Function(an-arg, another-arg)
```

The value of an item defined in a *Definitions* construct is always a future if it is possible to evaluate it as a future.

4.3.3. Bags

One abstraction suited particularly to the parallel mode of execution of Poligon programs is the *Bag* data type. Bags are implemented in Poligon so that they are formed as the result of efficient parallel operations and can be processed in parallel efficiently. Even when the elements of Bags are processed serially they perform efficiently. The lack of a defined ordering in the Bag means that the system can always return the first satisfied Future out of a Bag of Futures, causing minimum waiting for values. Similarly, when a program attempts to extract an element from a bag and there are no satisfied elements the process in which this happens will go to sleep until the next available future is satisfied.

A Bag is generated, for instance, as the value of the following expression. It is a Bag, which contains all of the *Words*, whose *Sound* is "*phoo*"².

¹Suitable *Force* operations are provided so that the time of evaluation can be controlled by the program if necessary. These force operators allow the program to perform *Eager Evaluation* if it is needed.

²The expression "Element • Sound" denotes extracting one of the values associated with the "Sound" field of the potential element in the bag. "•" is an operator that selects which of the values associated with the field is to be delivered.

4.4. Parallelising Abstractions

Poligon supports data representations which are designed to give the user a high level handle on the exploitation of parallelism. Most values computed in Poligon are derived as *Futures*. Computation is decoupled from the expressions which reference values. Futures are, however, completely invisible to the user in Poligon. It understands which functions are strict in their arguments and so waits for the satisfaction of a Future only when it is required. The programmer can, of course, declare his own non-strict functions and operators. All *DeFuturing* coercions are performed automatically by the Poligon system. Thus the following expression will deliver a list with two elements, one of which is the value of a and one of which is the sum of b and c . The first will be a future, if a is. The second will be the DeFutured value $b+c$.

```
List(a, b+c)
```

The efficient use of the bandwidth of the processor interconnection network is enhanced by the use of *Broadcast* and *Multicast* operations. Broadcast messages allow messages to be sent to every node in the system in a single operation. Multicast messages allow messages to be sent to a collection of nodes in a single operation. The Poligon system uses these extensively in the processing of the Bag data type and in the execution of groups of actions in parallel. It uses the same mechanisms to provide an efficient implementation for searching a collection of nodes on the blackboard for patterns, which tends to cause significant slowing of serial implementations because of the combinatorial nature of such searches. It allows the blackboard to be searched for bags of matching nodes in a single, fast operation. This provides a significant improvement over the serial construction of such collections.

4.5. Real-time processing

Real-time processing brings its own problems. Poligon provides a simple and regular mechanism for defining the interface between the Poligon system and its signal data. This data can be from an arbitrary number of different types of sources and is posted on the blackboard asynchronously.

Poligon also provides a mechanism by which each datum is timestamped from the time that it enters the system. These timestamps are propagated automatically by the system so that it is trivial for the programmer to manipulate time-ordered collections of values. This mechanism is required because the conventional implicit time ordering of data in lists cannot apply here and the non-ordered nature of Bags is sometimes not sufficient.

4.6. The control of assignment

Assignment is something which is likely to cause significant problems in any parallel system. Poligon constrains assignment in a number of ways. Side-effects are only permitted on the fields of nodes. All side-effects can be monitored by rules that might be interested in the changes to values. This removes the possibility of the knowledge base getting confused because of surgical side-effects to data structures at arbitrary times and at arbitrary places in the processor network. Assignment is also constrained so that all of the updates to the fields of a given node are done atomically, before any rules which might be triggered by these changes are allowed to trigger. Such atomicity helps to preserve the consistency of the system.

An example of a collection of updates to fields of a given node is given below. In this example the node *an-instance-of-words* is having two of its fields updated; *Sound* and *Letters*. Operators, such as " \leftarrow ", allow different sorts of modifications to be made to fields. Such operations might be "add this value to the values in this field" or "replace all of the values in the field". This avoids complex and potentially expensive expressions in the old value of the field being evaluated non-locally.

```
Change Type      : Update
Updated Node     : an-instance-of-words
Updated Fields   : Sound   $\leftarrow$  "phoo"
                  Letters  $\leftarrow$  [ f o o ]
```

5. Conclusions

This paper has described Polygon, a language and system for the investigation of problem solving on distributed-memory, parallel hardware. The language was described in the context of related work in the field and in terms of the abstraction mechanisms provided. No significant description of the underlying run-time support has been given.

The Polygon system is still young. Only recently have applications been mounted on it in earnest. Two distinct applications in the field of real-time signal processing are now being implemented and more applications are likely to be started in the near future. Polygon has proved to be well suited to these applications as far as they have gone. No results from the simulation process regarding the performance of Polygon programs are yet available. Significant problems have been found in the simulation of the fine-grained parallelism required by the Polygon metaphor. Such simulations are very time consuming, prone to bugs in the underlying system software and simulator, and are difficult to debug. It is for these reasons that Polygon also has a serial version, Oligon, which accurately emulates the behaviour of the parallel system but without true parallelism. A simulated processor array of 256 processors has recently been made available to the users of Polygon. This simulation will allow more satisfactory investigation of the properties of Polygon programs in the future.

6. Further Reading

For a significantly more detailed treatment of the Polygon language and system the reader is encouraged to consult [Rice86].

The following topics were not described or discussed but are relevant to the work described above. The reader is encouraged to consult the following for further information;

[KSL 85] for a description of the Advanced Architectures Project of which Polygon is a part.

[Delagi 86] for a description of CARE, the hardware simulator used by Polygon, and of the particular hardware being simulated.

[Schoen 86] for a description of CAOS, the concurrent object oriented system running on the CARE machine, which Polygon uses as its operating system.

[Ensor 85], [Lesser 83], [Aiello 86] and [Fennel 75] for other approaches to parallel problem solving using blackboard systems.

7. References

- [Aiello 86] Nelleke Aiello. *User-Directed Control of Parallelism; The CAGE System*. Technical Report KSL-86-31, Heuristic Programming Project, Computer Science Department, Stanford University, 1986. Also in Proceedings of DARPA Expert Systems Workshop, April 1986, and [Stanford 88].
- [Davies 86] Davies, Byron. *CAREL: A Visible Distributed Lisp*. Technical Report KSL-86-14, Heuristic Programming Project, Computer Science Department, Stanford University, 1986. Also in Proceedings of DARPA Expert Systems Workshop, April 1986.
- [Davis 77] Davis, R. and King, J. *An Overview of Production Systems*. In E.W. Elock and D. Michie (eds), *Machine Intelligence 8: Machine Representation of Knowledge*, John Wiley, New York 1977.
- [Delagi 88] Bruce A. Delagi, Nakul P. Saraiya, Gregory T. Byrd and Sayuri Nishimura. *CARE User's Manual*. Technical Report KSL-88-53, Heuristic Programming Project, Computer Science Department, Stanford University, 1988.
- [Ensor 85] J. Robert Ensor and John D. Gabbe. *Transactional Blackboards*. Proceedings of the 9th International Joint Conference on Artificial Intelligence: 340-344, 1985.
- [Fennell 77] Richard D. Fennell and Victor R. Lesser. *Parallelism in AI Problem Solving: A case Study of Hearsay-II*. IEEE Transactions on Computers: 98-111, February, 1977.
- [Gabriel 84] Gabriel, Richard P. and McCarthy, John. *Queue-based Multi-processing Lisp*. Proceedings of the ACM Symposium on Lisp and Functional Programming: 25-44, August, 1984
- [Halstead 84] Halstead, Robert H. Jr. *Implementation of Multilisp: Lisp on a Multiprocessor*. Proceedings of ACM Symposium on Lisp and Functional Programming: 9-17. August 1984.
- [Hayes-Roth 85] Barbara Hayes Roth. *Blackboard Architecture for Control*. Journal of Artificial Intelligence. 26: 251-321, 1985.
- [Hewitt 73] Hewitt, C., P. Bishop and R. Steiger. *A Universal, Modular Actor Formalism for Artificial Intelligence*. Proceedings of the 3rd International Joint Conference on Artificial Intelligence: 235-245, 1973.
- [KSL 85] Knowledge Systems Laboratory. *Knowledge Systems Laboratory 85, incorporating the Heuristic Programming Project*. KSL, Dept. of Computer Science, Stanford University, 1985.
- [Lesser 83] Victor R. Lesser and Daniel D. Corkill. *The Distributed Vehicle Monitoring Testbed: A Tools for the Investigation of Distributed Problem Solving Networks*. The AI Magazine, Fall:15-33, 1983.

- [Nii 79] H. Penny Nii and Nelleke Aiello. *AGE: A Knowledge-based Program for Building Knowledge-based Programs*. Proceedings of the 6th International Joint Conference on Artificial Intelligence: 645-655, 1979.
- [Nii 86] H. Penny Nii. *Blackboard Systems*. Technical Report KSL-86-18, Knowledge Systems Laboratory, Computer Science Department, Stanford University, April 1986. Also in AI Magazine, vol. 7-2 and vol. 7-3, 1986.
- [Rice 84] James Rice. *The MXA user's and writer's companion*. Systems Programming Ltd., The Charter Abingdon, Oxon, UK. 1984.
- [Rice 86] James Rice. *The Poligon User's Manual*. Technical Report KSL-86-10, Heuristic Programming Project, Computer Science Department, Stanford University, 1986.
- [Schoen 86] Eric Schoen. *The CAOS System*. Technical Report KSL-86-22, Heuristic Programming Project, Computer Science Department, Stanford University, 1986. Also in Proceedings of DARPA Expert Systems Workshop, April 1986.

The Poligon User Manual

by
James Rice
(Rice@Sumex-AIM.Stanford.Edu)

**Knowledge Systems Laboratory
Stanford University
701 Welch Road
Palo Alto, CA 94304**

The author gratefully acknowledges the support of the following funding agencies for this project; DARPA/RADC, under contract F30602-85-C-0012; NASA, under contract number NCC 2-220; Boeing Computer Services, under contract number W-266875.

1. Context for this document

1.1. Intended Reader

This document is aimed at the new user of the Poligon language and Poligon system. It is assumed that the reader will be familiar with the operation of Texas Instruments Explorer™ lisp machines¹ and with Common Lisp itself. Familiarity with AGE would be useful in understanding the nomenclature used in Poligon but it is by no means essential.

1.2. About this document

It is hoped that this document will be both helpful and meaningful to both the prospective Poligon user, the experienced user, who is in need of a reference manual and to readers, who have no intention of doing anything other than finding out about Poligon. These are in many ways conflicting goals, since the detail necessary for a reference manual can often be too great for the casual browser. An attempt has been made to structure this manual in such a way as to allow the reader to skip over excessive detail. To assist both the novice and experienced user the manual is equipped with a sizable index.

Within the text of this manual great use is made of *italicized* script. Italic text is used to denote three things; either significant words or phrases in the system, places where some piece of text is being *stressed* or quoted sections of program examples or descriptions. Thus if the reader was to find the following program example

If : *a-value* > another-value

then *a-value* would be referenced in italics because it is a reference to the example above. The fact that this is an example of an *If Part* of a rule would also be italicized as it has been here to show which terms have special significance or meaning.

The reader should find that all terms which are in bold script and all terms which are in italic script, which refer to significant terms are indexed.

1.3. Definition of Terms

In this document the following terms will be held to have the associated meanings:

DeFuturing	That coercion which extracts the value from a <i>Future</i> , waiting for that value to be evaluated if necessary.
Evaluation	In this document the term <i>Evaluation</i> will, in all cases, refer to the process of executing the code needed to determine the value of the relevant object. All code in the Poligon system is compiled through the Common Lisp compiler. There is no code, which is <i>EVAl</i> ed. It is conspicuously the case, therefore, that free references will be determined on the basis of their lexical, not on the basis their dynamic environment. Thus <i>Lazy Evaluation</i> refers to the lazy execution of a call to a closure in which resides the code associated with the de-

¹Explorer is a trade mark of Texas Instruments Corporation.

	termination of a value, not the lazy calling of <i>EVAL</i> on a quoted list. This makes the semantics of the Poligon system much clearer.
Field	This refers to a named data item within a Node. It is considered in all ways synonymous with the term <i>Slot</i> .
:Keyword	This will denote a symbol which is interned in the keyword package, that is to say a symbol, whose slashified denotation is that of a symbol preceded by a colon.
Keyword	This refers to a language <i>Keyword</i> in the Poligon [or L100] language. A <i>Keyword</i> in these languages is much like a <i>Keyword</i> in a language such as Pascal. "If" is an example.
Latest	This word is used in the AGE sense, which is to say, it refers to the first element in the <i>Value list</i> associated with the <i>Fields</i> of a Node. Because of the nature of Poligon systems it is by no means necessarily the case that this denotes the most recent value with respect to the processing of the data over time. If, however, the <i>Field</i> which is being examined has a sorting key then this will indicate the most recent/greatest element.
Model	The user defined application, which runs within the Poligon <i>Blackboard</i> framework. More specifically, this includes all code written in the Poligon language and any external functions which the user has chosen to write in Lisp but excludes all that is provided by the Poligon system.
Modify	This term is used in a very general sense, as opposed to the specific AGE sense (\$Modify), in which it meant adding a new element to the front of the <i>Value List</i> . In Poligon this could mean any user definable modification operation.
Node	An element in the solution space and, more generally, any object on the Poligon <i>Blackboard</i> . <i>Nodes</i> are the data structures on which <i>Rules</i> operate.
Rule	Rules are the program constructs in which the user encodes procedural or declarative knowledge. The consist of condition/action pairs that are sensitive to the state of the evolving solution and operate on that solution.
Slot	This refers to a named data item within a Node. It is considered in all ways synonymous with the term <i>Field</i> .
Special	This term is never used in the Common Lisp sense. There are no dynamically bound variables in Poligon <i>Models</i> . <i>Special</i> is simply used as an adjective, which denotes something of significance.
Value List	This refers to something akin to the AGE concept of a history. It is a list of values associated with a <i>Field</i> in a Node. In Poligon the structure of the value of a <i>Field</i> can be considerably more complex than simply a list (see Section 3.3.4).

2. Introduction

The Poligon system is an attempt at producing an application independent *Blackboard* framework, which is able to exploit parallelism inherent in the application by being inherently as independent of implicit sequential dependencies as possible and by executing the suitably compiled user code on an, as yet simulated, array of processors.

The architecture of Poligon, at least from the user's point of view, is independent of the nature of the target processors and their topology. It is very much the policy of the design of Poligon that the interface provided to the user should be entirely decoupled from the target architecture and, wherever possible from the explicit expression of parallelism. This is for the obvious software engineering reasons of wanting to produce reliable, maintainable, debuggable and portable code, quickly. This, of course, puts much more reliance on the underlying system to make a reasonable attempt at exploiting the parallelism inherent in the system. It is assumed that this is the job of machines. Just as no sensible person would voluntarily write in assembler, so the programmer should not be forced to write hard-coded implementations of his ideas. It should be noted, however, that exceptions to this policy do exist and are mentioned in Appendix P, the appendix, which concerns user defined parallelism.

The Poligon system consists of two distinct components; the *Poligon Compiler* and the *Poligon Run-Time System*. These components are linked only in that the compiled code from the *Poligon Compiler* runs in the context of the *Poligon Run-Time System*, and that both are loaded when Poligon is loaded.

Poligon has two modes of operation; a *Serial* mode and a *Parallel* mode.¹ Considerable effort has been made to make the semantics of these two modes as similar as is reasonably possible. The *Serial* mode exists primarily as a development tool. It is vastly faster than the simulated *Parallel* mode and it is therefore hoped that this will be a useful development tool even if only because of its speed. Luckily, however, the *Serial* mode is also more intelligible, when it comes to debugging. Because of the similarity of these modes they will not be referred to as distinct except for where there are material differences in either their semantics, operation procedures or user interface.

The following sections of this introduction give a brief introduction to the Poligon architecture. Section 3 of this document is a description of the Poligon language and the *Compiler*. Section 4 is a description of the *Poligon Run-Time System*. Numerous appendices follow these sections detailing different parts of the Poligon system.

2.1. The Poligon Architecture

Poligon takes as its starting point AGE. That is to say it is a system which implements the concept of *Nodes* - record-like data structures - on a *Blackboard* and these *Nodes* are acted on by rules, which are chunks of knowledge represented as condition-action pairs. From

¹The terms *Serial* and *Parallel* here are somewhat misleading and exist for historical reasons only. The *Parallel* mode in Parallel because it uses the CARE simulator to simulate parallelism. This is still executed on a uniprocessor, so is really a serial system, but its semantics are such that they would have been the same if the program were running on a real multiprocessor. The *Serial* mode is a mode in which the CARE simulator is emulated. In practice, therefore, the semantics are very similar to the *Parallel* mode. The major difference is that the CARE system is more likely to be deterministic and a considerable amount of instrumentation is provided to monitor the behavior of the system. The *Serial* mode has none of this instrumentation but is vastly faster.

this point on it diverges from AGE. It should be noted, however, that there are two quite distinct "Views" of the Poligon architecture. One is the view of the user of the system, the other is the view inside the system. A number of features in the system have been designed to be in some sense compatible with the likely target architecture, which is expected to be an array of distributed memory MIMD (Multiple Instruction stream, Multiple Data stream) processors, which are able to communicate with one another by message passing. It is assumed that there is no global data, and hence there are no shared variables. It is also assumed that there will be a large number of these processors, comparable to the number of active *Nodes* in the system, not of the order of tens. The architecture of Poligon by no means requires these assumptions. It is simply in some senses "optimized" for such architectures.

Because of the uncertainty of the target machine and the low level nature of many of these aspects the user has been to the greatest degree possible detached from them. The user perceives a hierarchically structured *Blackboard* system whose rules are entirely daemon driven. Global variables are not implemented and the sharing of data is achieved by associating it with the level in the hierarchy of the *Blackboard* which is directly visible to all those *Nodes* that might want to see it.

What follows is a brief discussion of the architecture of Poligon from a lower level. This is provided for the interest of the user who might want to know some of the motivations for the features of Poligon. Reading it is by no means necessary in order to use Poligon, though, sadly, it may be of use to allow efficient use of Poligon.

2.1.1. Inside Poligon

AGE, like most *Blackboard* systems has the concept of a *Blackboard*, which is a globally accessible database, and a distinct *Knowledge Base*, consisting of *Knowledge Sources*, which are themselves divided into smaller chunks of knowledge; Rules.

Poligon takes as its underlying metaphor the concept of a *Blackboard Node* as a *process*. In the most extreme view of this a Node would be a processor but this is by no means required if the processors in the network are capable of multi-tasking. This view of *Nodes* has the effect that the *Nodes*, instead of being passive entities in the conventional *Blackboard* metaphor, are active entities. From this point of view it requires no great leap of the imagination to envisage the distribution of the rules of the system over the processor network so that the *Nodes* have the rules necessary for them intimately associated with them. At this point the concept of a distinct database and *Knowledge Base* disappears, at least at the implementation level.

AGE uses a simple scheduling mechanism. A global stack/queue of event tokens was kept and events were taken from the stack in order to activate *Knowledge Sources*, which might cause more events to be posted on the event list. Control was therefore passed around explicitly between the *Knowledge Sources* by the agency of this global list.

In Poligon there are no global variables. This means that a shared data structure, such as a global event queue, is neither possible nor desirable, since shared data structures are likely points for serialization. This restriction was made because an arbitrary processor might have to communicate over an arbitrary distance across the processor network in order to see such a global queue. This would cause hot spots in the grid and major consistency problems. In discarding this global control mechanism Poligon needed another way to cause the activation of rules. Clearly all of the rules could attempt to fire all of the time. This could cause the hardware to be very busy doing work waiting for rules to fire. This work, since it might require non-local references, could well cause significant degradation of the

performance of the system. Polygon, therefore, opts for a distributed event-based mechanism for rule activation. Rules are associated with *Fields of Nodes* and "watch" them. When a suitable event such as an update, happens to the *Field* being watched, the rules that are doing the watching are triggered. The Node that owns this *Field* therefore becomes, in a loose sense, the Polygon equivalent of AGE's *focus-node*.

Thus for each rule the triggering *Field* has to be specified. This allows the system to place the rules for the specified classes of Node and the specified *Field* around the processor grid in a manner appropriate to the application. This means that the AGE concept of the *Knowledge Source* as the fundamental unit of scheduling no longer applies. Not only can the rules of a *Knowledge Source* fire quite independently of one another but they may also be distributed over quite distinct parts of the processor network. Thus the term *Knowledge Source* should not be taken to represent quite the same sort of entity as it does in AGE. It is merely a knowledge representation abstraction.

Polygon is aimed amongst other things at investigating real-time signal processing problems. This by no means precludes other applications, of course. It simply means that the system has a real-time clock, some special mechanisms designed to cope with real-time problems and requires all data coming into the system to be time-stamped. All input data for the system is fed through a procedure, which is user defined and is expected to put the data on the *Blackboard* in a suitable fashion. This data is spread around the network and acted on automatically by a number of input processing *Nodes*. This all happens in a manner about which the user need not worry.

3. The Poligon Language, Its Syntax and Semantics

The Poligon language is the source language in which rules are written so that they will run under the Poligon system. The language is intended to provide a common front end for the user so that the rules developed for the system should work transparently between the *Serial* and *Parallel* modes. This removes the need to have multiple sets of source code, which need to be updated.

Apologies are made now for any forward references in this document. It is the nature of language descriptions that in order to describe a language in a suitably structured and top-down manner forward references must be made.

The Poligon *Compiler* and the Poligon *Run-Time System* can be loaded up by executing the form:

```
(Make-System 'Poligon :Silent :Noconfirm :Nowarn)
```

The Poligon language is built on top of the L100 language, much as flavors are built up from basic components. The Poligon language therefore can be thought of as an extension to the behavior of the L100 language with a number of modifications. Of the modifications the only ones that the user is likely to see are the addition of a few new types of basic value. These are documented below in Section 3.3.3.

The language extensions themselves define four distinct parts of the user's input to the Poligon system. These are;

- The Class Declarations
- The Data Input
- The User Defined Initialisation
- The Knowledge Base itself, which is encapsulated in *Knowledge Sources* and rules.

These will be described in turn. The grammar is shown in Appendix J.

3.1. Poligon and L100

The Poligon language was defined by building it on top of the L100 language. In this section, therefore, we will first describe the L100 language and then the extra features that were built on top of it to give the Poligon language. The description will be done in this order so as to give the reader a feel for the syntax of the language before any complications to do with concurrent problem-solving are added. L100 is a simple language which compiles into Common Lisp. Its semantics are largely those of Common Lisp and so it exists mainly to provide a syntactic infrastructure on which other languages can be built.

3.2. Introduction

L100 is a programming language. It is quite powerful in its own right but its major features are its simplicity and its extensibility. The language is intended to achieve different goals from those of Lisp. It is intended to be easy to read and, at least to some extent, similar to many of the other languages that exist, which have infix *Operators* and *Keywords*.

L100 itself is in fact an application written on top of a generic parser tool. L100 is that language, which was written in order to express the code for the parser tool in a manner that could be bootstrapped easily. Thus L100 is just one of the possible uses of this parser tool that reflects the aesthetic inclinations of its author. The generic parser tool is thoroughly documented in its source code. It is a parser interpreter, as opposed to a parser generator. It is a top-down parser with backtracking. It is, therefore, not constrained by restrictions on the source grammar such as *LL1*. It will match source code against a *Grammar* requiring arbitrary backtracking, though if the compiler writer chooses to implement the semantics of the compiler by the use of side effects then the backtracking might not generate the correct code.

The remainder of this section comes in three main components. These cover; how to load up the language, the language in outline including the concept of *Operators* in L100, the way in which L100 handles *Lists* and the implementation of the L100 *Grammar*. An appendix gives some examples of simple programs written in L100.

3.2.1. An Outline of L100

L100 is a block structured language with closed constructs. It is therefore different from languages such as Pascal in as much as it has no Syntactic concept of *Begin* and *End*. This is, of course, only of aesthetic significance. Constructs in the language are, by convention, closed by keywords, which have the prefix of *End*. Thus *If* is closed by *EndIf* and *Let* is closed by *EndLet*.

L100 is case insensitive, like Common Lisp. L100 is, like Lisp and Pascal, layout independent, though it does sometimes rely on whitespace for its lexis. These cases are described below, but the corollary is that you can always put any number of *Comments* or whitespace characters between tokens, but tokens may not have any whitespace within them. *Strings* are a special case and are mentioned below.

"Statements" are separated by the *Lozenge* symbol (\diamond) or bang symbol (!). This is intended to be a noisy way of highlighting that side effects are being made.

3.2.2. L100 lexis rules

The L100 language has different lexis rules from Lisp. This is so as to allow special symbols to have syntactic meaning. This means that the lexis rules are much like those of Pascal-like languages. *Identifiers* can only begin with letters, & signs or dollars, *Numbers* can only start with a numeral and so on. There are a couple of things that should be noted however. *Operators* that start with an *Operator* delimiting character (see below) can only contain characters that are reserved for *Operators*. The lexis rules are relaxed for *:Keywords* (i.e. tokens that begin with a colon). Thus anything that follows a colon, up until the next whitespace character or parenthesis will be read by the reader as a *:Keyword*. The same can be said of quoted items. In both of these cases it is imperative that a whitespace character should be put between these symbols and commas in *Parameter Lists*, otherwise the commas will be swallowed into the symbol and a syntax error will result.

There are a few *:Keywords* used by the lexical analyzer itself to denote types of tokens. These are considered special by the lexical analyzer and must be quoted before they can be read as tokens. It is unlikely that any user code will contain any of these. The *:Keywords* are shown in Table 3—1.

:Start_of_file	:End_of_file	:Bra
:Ket	:Comma	:Slash
:Lsq	:Rsq	:Colon

Table 3—1 L100 Reserved :Keywords

There are a set of characters that cannot be used in the middle of *Identifiers*, since they are used as *Operators*, syntax delimiters or are reserved for the user to use as *Operators*. These are shown in Table 3—2.

≡	≠	=	<	>	→	+	*	/
()	,	∅	•	!	[]	^
@	⊃	↔	~	?	≤	≥	←	%
↑	↓	⊕						

Table 3—2 L100 Reserved Characters

Strings are conspicuously different in L100 from Lisp. *Strings* in Lisp suffer from the problem that you cannot stretch the source code representation of a *String* over multiple lines without embedding carriage control in the *String*, which in turn gets compiled into the *String*. An unpleasant consequence of this is that *Strings* which are not closed properly can cause large quantities of code to be swallowed. This is not the case in L100. No token is allowed to cross a newline boundary. This means that a *String* that is not closed on a line will cause a lexis error. If the user wants *Strings* that are longer than the amount of space that he is prepared to allocate on a given line *Strings* can be continued by the use of the infix literal *String* concatenation *Operator* "&". Thus an example *String* might be as follows:

```
"This is a very long "&
"string, which is spread "&
"out the way I want and "&
"split over a number of "&
"lines."
```

"It should be noted that the "&" *Operator* is executed at compile time, not at run time. Thus it is able to provide constant documentation *Strings* but cannot be used as a general *String-Append Operator*. There is, of course, nothing to stop the user from defining such an *Operator*.

Comments are defined in two ways in L100. The first is the "end of line" type of *Comment* supported by Common Lisp and introduced by a semicolon. The second is a bracketed *Comment* construct, which allows *Comments* in the middle of code on a line. A bracketed *Comment* may not stretch over a line boundary. This is to prevent code from being swallowed in *Comments*. The bracketed *Comment* is denoted by the "{}"; characters. Thus the following is a legal piece of L100 code

```
a•length {the length of a} + 3 →
new-length {new length} ∅ ; Comment
```

3.2.3. The L100 predefined Operators

L100 comes equipped with a number of predefined *Operators*. These are defined as shown in Table 3—3.

Operator	Lisp Equivalent	Precedence	Associativity
<	<	15	Left
>	>	15	Left
≤	<=	15	Left
≥	>=	15	Left
+	+	20	Left
-	-	20	Left
*	*	25	Left
/	Quotient	25	Left
≠	Not_equal	15	Left
⊃	Cons	25	Left
↔	Append	20	Left
→	Setf	2	Left
=	Equal	15	Left
•	<Application>	30	Left
Or	Or	5	Right
And	And	10	Right

Table 3—3 L100 Language Operators

These have been chosen to give the sort of effect that one would expect in a language with infix Operators.

Not_Equal is defined as (*Not (Equal x y)*).

Since these Operators are not easily printable on standard printing devices without special text processing an extra set of Operators has been defined which provides an easily printable representation of the source code at, perhaps, the cost of a little legibility on Lisp machines. These are shown with their equivalents in Table 3—4.

Operator	Easily Printable
←	<-
≡	==
↔	<>
≠	%=
⊃	~
•	@
→	->
≤	<=
≥	>=

Table 3—4 L100 Operators and their printable equivalents

It should be noted that the normal mode of use of operators in L100 is in the *infix* form. This is, however, not mandatory. Operators can be used in a prefix form if required. If this is done then the arguments provided are used as specified. No reordering is performed as it might be in the case of the infix form. Thus both of the following expressions are legal L100 and will have the obvious effect.

a ↔ b	(Append a b)
↔(a, b, c)	(Append a b c)

3.2.4. Defining New Operators

One of the features of significance in L100 is not simply the presence of infix *Operators* but the ability of the user to define his own and to modify the behavior of the existing *Operators*. The predefined *Operators* are only provided as a convenience to the user and are in no way a structural feature of the language. Defining *Operators*, therefore, is one of the key mechanisms in L100 by which the language can be tailored to suit the application at hand.

Operators are defined by the use of the *Declare_Operator* procedure. This procedure is defined as follows:

```
Declare_Operator (Name, Function_to_call, Precedence,  
                  Associativity, Reverse_the_arguments,  
                  Omit_the_operator, Language)
```

The use of this procedure is fully documented in the source code. A couple of examples of its use, however, are given here to show how easy it is:

```
Declare_Operator('→ , 'setf , 2, :Left , t, nil, :L100)◇
```

This is the L100 source code necessary to define the assignment *Operator* in L100. The *Operator* is called "→" and instances of this *Operator* are compiled into calls to the *Setf* procedure. The *Operator* has a precedence of 2 (this is very low) and it is left associative. The operands are reversed in the generated code and the *Operator* is included, i.e. this is not a function application *Operator*. Thus the following transformation happens at compile time

a → b

compiles into

(Setf b a)

The second example is of the L100 infix function application *Operator*. This is declared with the following code:

```
Declare_Operator('• , nil, 30, :Left , t, t, :L100)◇
```

This *Operator* is called "•" and there is no Lisp function associated with it. It has a high precedence of 30, is left associative, has its arguments reversed and has the (null) function omitted in the generated code. Thus the following transformation occurs at compile time:

a • b

compiles into

(b a)

Clearly, then, the following:

a • b → c

compiles into

(Setf c (b a))

and

$c \rightarrow a \bullet b$

compiles into

(Setf (b a) c)

3.2.5. L100 and lists

Lists in L100 are different from *Lists* in Lisp in that *Lists* are always treated as representations of data structures. They are not used to denote the language itself in any way. Thus arguments to a function are not expressed as a *List*, though an argument may be a *List*.

Lists in L100 are enclosed in brackets and are implicitly "backquoted". Thus the Lisp list

'(a b c)

would be expressed as follows:

[a b c]

The *Comma* concept associated with backquoted *Lists* in Common Lisp can be expressed in three ways in L100. The direct equivalent of a *Comma* is the *Uparrow*. Thus the Lisp expression:

`(a b , c d)

would be expressed as follows:

[a b ↑ c d]

The *Comma At-sign* construct is mirrored by the *Downarrow*. Thus the Lisp expression:

`(a b , @ c d)

would be expressed as follows:

[a b ↓ c d]

There is a third way of expressing the positioning of evaluated elements in a *List*. This is by the use of the *Percent* delimiters. Within a percented component of a *List* all of the arguments are evaluated. This is a shorthand for a lot of *Uparrows*. Thus the Lisp expression:

`(a b , c , d , e , f , g h i j)

can be expressed as follows

[a b % c, d, e, f, g % i j]

Of course a call to the function *List* can always be made as well.

As is the case for *Operators*, the special symbols in the *List* notation can be substituted by a more easily printable form. These are shown in Table 3—5.

Normal	Easily Printable
↑	^
↓	?

Table 3—5 *List specifier characters and their easily printed representations*

3.2.6. Defining extensions to the grammar

Other than defining your own *Operators*, L100 provides the user with the ability to define his own extensions to the *Grammar*. It is by such mechanisms that one produces specializations of L100 to produce languages such as the Poligon and Cage languages.

The *Grammar* can be extended by two mechanisms. It can be extended statically by the use of the *Converter* procedure, which takes a *List* of filenames and a *:Keyword* denoting the name of the language and converts the *Grammar* into a form that the compiler can understand.

It can also be extended incrementally by the use of the procedure *Declare Production*, which takes three arguments; a *String* denoting the production line in the *Grammar*, a *String* denoting the action line in the *Grammar* and a *:Keyword* denoting the name of the language.

The definition of new *Grammar* constructs (productions) on the fly is a dangerous and difficult process so the user is advised to read all of the documentation associated with the parser tool and the *Grammar Converter* before he tries.

3.2.7. L100 and Binding

L100, like Common Lisp, is a lexically scoped language. L100, however, is much more strict about *Binding* than Common Lisp. In L100 it is not possible to achieve *Dynamic Binding* unless it is explicitly asked for. *Lambda Binding* is not allowed at all. Thus the following L100 and Common Lisp programs are *not* equivalent.

```

Variable a-global-variable ← 42

Define a-function (a-global-variable)
  a-global-variable•another-function
EndDefine

Define another-function (an-argument)
  print (a-global-variable)
  print (an-argument)
EndDefine

Do 100•a-function

and

(DefConst a-global-variable 42)

(defun a-function (a-global-variable)

```



```

        (another-function a-global-variable)
    )

    (defun another-function (an-argument)
      (print a-global-variable)
      (print an-argument)
    )

    (a-function 100)

```

In the first case the L100 program will print out 42 and then 100. In the Common Lisp case the program will print out 100 and then 100. Similarly the following programs are not equivalent.

```

Variable a-global-variable ← 42
Define a-function (an-argument)
  Let a-global-variable = an-argument
  In an-argument • another-function
EndLet
EndDefine

Define another-function (an-argument)
  print(a-global-variable)
  print(an-argument)
EndDefine

Do 100 • a-function

```

and

```

(DefConst a-global-variable 42)

(defun a-function (an-argument)
  (let ((a-global-variable an-argument))
    (another-function an-argument)
  )
)

(defun another-function (an-argument)
  (print a-global-variable)
  (print an-argument)
)

(a-function 100)

```

The same results are produced by these programs as were produced by the preceding examples. Thus the L100 *Let* construct achieves only *Lexical Definition* and not *Dynamic Binding*. This is also the case with function arguments; *Lambda Binding*.

There is, however a need for *Dynamic Binding*, since the underlying Common Lisp system requires it for such global symbols as *Terminal-IO*. L100, therefore provides a *Bind*

primitive to achieve this. The following L100 program is equivalent to the last Common Lisp example.¹

```
Variable a-global-variable ← 42

Define a-function (an-argument)
  Bind a-global-variable = an-argument
  In an-argument • another-function
  EndBind
EndDefine

Define another-function (an-argument)
  print (a-global-variable) ◊
  print (an-argument)
EndDefine

Do 100 • a-function
```

3.2.8. L100, Type Checking and Pragmata

L100 supports no static type checking. It does, however, support the specification of dynamic type checking. This is done in the *Parameter Lists* for functions. The types that an argument can take on are defined after the argument. A set of types can be specified. Thus a function could be defined as follows:

```
Define a-function
  (an-argument,
   an-integer-or-a-string : Integer : String,
   another-argument)
  ...
EndDefine
```

This function takes three arguments. The first and third arguments can be of any type. The second is allowed to be either an integer or a string. Failure to conform to either of these types will cause an error at run-time. The types that can be specified are any symbolic types that can be given to *typep*. Note these are *Identifiers* delimited by *Colons*, not *:Keywords*. The specification of the *: Integer* type is the same as saying that the argument should conform to the specification of (*typep an-integer-or-a-string 'Integer*).

L100 also supports a mechanism which allows *Pragmata* to be defined to act upon function arguments. There is only one such *Pragma* in L100, called *Lazy*, but languages derived from L100 might well define more for themselves. All such *Pragmata* are defined using the same syntax as that for type checking. The *Lazy Pragma* defines arguments to be lazily evaluated. There is a strict declare-before-use requirement for functions with *Lazy* arguments. The following is an example of code in which *Lazy* arguments to functions are used.

```
Define a-function (a : lazy, b)
  Princ("Hello") ◊
  if b then a endif
```

¹Note: The Polygon model of parallelism does not allow dynamic binding. If the user chooses to use dynamic binding then this will not be strictly in accordance with the programming model i.e. a true Polygon machine might not support such a program.

```
EndDefine
```

```
Define another-function (c, d)
  Princ(d) 0
  a-function(Princ(" there.  "), c)
EndDefine
```

```
Do another-function (t, "Example 1: ")
```

```
Do another-function (nil, "Example 2: ")
```

In this example the function *a-function* is defined to have a lazy argument, called *a*. If the argument *b* is non-nil then *a* will be evaluated. If not then *a* will not be evaluated. Thus the output from this program will be "Example 1: Hello there. Example 2: Hello".

3.2.9. The Outer-most Level of the Language

An L100 program consists of a number of outer most level declarations. Most of the *Grammar* and complexity of the language is associated with the definition of expressions, which can be put within these outermost level declarations. The outer most level declarations do, however deserve a little comment. They are defined by the production: *Toplevelstatement* and can consist of a number of items.

3.2.9.1. Constant and Variable declarations

Constants are declared by the following form:

```
Constant a-constant-name = an-expression
```

This is equivalent to a *DefConstant* declaration. The expression is evaluated at load time.

Variables are declared as follows:¹

```
Variable a-variable-name ← an-expression
```

This is equivalent to a *DefParameter* declaration. The expression, like that for *Constants* is evaluated at load time. The initialization is mandatory.

3.2.9.2. Structure Declarations

Structure types can be defined in L100. The following is an example of such a declaration.

```
Structure structure_name
  Fields field_1, field_2, field_3
```

This declares a *Structure* called *structure_name*, which has three fields, which are accessed by the functions *field_1*, *field_2* and *field_3*. A type predicate is declared, whose name will be *is_a_structure_name* and a keywordless constructor will be generated called *Make_structure_name*. The prefixes "IS_A_" and "MAKE_" are defined as the values of

¹Note: Global Variables are not allowed in a Polygon application, since there is no global data concept. The ability to define global variables has not been removed from the Polygon language, however, because it is still useful to define globals that denote things that are not related directly to the functioning of the model itself, e.g. trace files and such which would not exist in a real system.

the constants *Make_Prefix* and *Is_a_Prefix*. Redefining these will allow the user to change this behavior if desired.

3.2.9.3. Function and Procedure declarations

These are of the obvious form and are covered in Appendix B. No distinction is made between functions and procedures in L100.

3.2.9.4. Outermost level statements

There is a need, sometimes, to express outermost command level side-effects. These are often assignments to system global variables, which are made at load time. This can be done by the use of the *Do* construct. The statement

```
Do 42 → System-Variable
```

will assign 42 to *System-Variable* at load time.

3.2.9.5. Compile time statements

L100 has provision for the user redefinition and/or extension of the compiler. There is a need, therefore, to allow the user to make these redefinitions at compile time, otherwise they would be done too late. The *Execute* construct is used for this purpose.

```
Execute Declare_Operator  
      ('an-op , 'a-function , 2, :Left, t, nil)
```

This will cause the *Operator An-Op* to be defined, so that the compiler knows about it, at compile time.

A peep-hole is allowed down into Lisp at this point, in case there is an extreme need to express something that is not supported in L100. This is the function *Literal*.

Literal takes a *String* argument and reads and evals from it. Thus the statement

```
Execute "<<Some strange and horrible bit of lisp>>"•Literal
```

will cause that bit of Lisp to be evaluated at compile time.

3.3. General Linguistic matters

This section relates to a number of issues that are significant concerning the Poligon language, either because they refer to changes to L100, features which are unique to the Poligon language or for which an understanding is necessary before the ensuing description of the language can be fully appreciated.

3.3.1. Packages

The Poligon system code is resident mainly in the *Poligon Package*. This package exports a number of symbols, which are used by the *Poligon-User* package. It is generally assumed that any Poligon model will be written in the *Poligon-User* package or a package which is built on it.

3.3.2. Data Types

There are three data types that might be of interest to Poligon model writers. These are *Bags*, *Sets* and *Lazy lists*. *Bags* are documented fully in Section 3.3.3.4, the section concerning searching the *Blackboard*. *Sets* are just like bags only they cannot contain any duplicate elements. They are not described as such but will be described in terms of their differences from *Bags* in Appendix E. *Sets* and *Bags* share the same operations. *Lazy Lists* are lists, which can be used by non-strict functions. They act just like lists but their *Tails* are defined in terms of value generating functions, not Cons cells. If the function *Tail* is called with a *Lazy List* as its argument it gradually evaluates the elements, which are needed. The list preserves *EQness*, though this should not be of significance unless the user chooses to ignore the advice given in Section 3.3.2.3, the section concerning equality. The functions for manipulating *Lazy Lists* are described in Appendix E, the appendix on data structures, as are those for *Bags*.

There are three data types which should not be visible to the Poligon model writer but which have an effect on the system. These are *Remote Addresses* and *Futures* and *Multi-Futures*.

3.3.2.1. Remote Addresses

This data type is an internal implementation detail of the *CARE* architecture. All *Nodes* in the system are seen as *Remote Addresses* in the processor net. This behavior is emulated in the *Serial* mode so that any time the user prints out, for instance, a *Blackboard* Node it will be printed out in the following form:

```
#<Remote xxx>
```

where *xxx* is the name of the Node. This may be ignored by the user, since the implementation hides any need for the user to deal with this data type. However if the user attempts to write any user functions that poke inside the Poligon implementation then he will find them all over the place. It cannot be stressed too strongly that the user should not resort to this practice. If you really need to do something that is not provided by the system then maybe other people may need it too and so it should be made part of the system and integrated properly. A major purpose of the *Compiler* and *Run-Time System* structure of the Poligon system is to protect the user from changes to the internal representation of the system.

An important feature of the fact that Poligon *Nodes* are seen by their *Remote Addresses* is that type comparison functions cannot be used on them usefully. For instance, the *Strict-Type-Of* #<Remote Sheep-42> would not be 'Sheep', it would be 'Remote-Address', independent of the type of the Node pointed to by the *Remote Address*.

3.3.2.2. Futures and Multi-Futures

Futures, and their close cousins *Multi-Futures* are values which represent *promises* for values. They are used extensively by the Poligon implementation. They are used because they allow the system, in principle, to carry on doing something else whilst a value is being evaluated by a different process. A process will always suspend itself if it needs the value of a *Future*, which has not been satisfied.

All values, which are extracted from the *Fields* of remote *Nodes* are represented as *Futures*. Thus most values that you are likely to want to represent or manipulate in Poligon are represented as or contain *Futures*.

Futures can be seen in the Poligon system when they are printed out because they will be represented as:

```
#<Future xxx>
```

where *xxx* is either the value that the *Future* represents or "Unsatisfied", an indication that the *Future* has not yet been satisfied.

The proliferation of *Futures* in Poligon might, at first sight seem to be likely to cause problems. For instance the following expression might seem to be an error at first sight:

```
#<Future 42> + #<Future 127>
```

where `#<Future xxx>` denotes an expression whose value is a *Future*. Even though the values of the *Futures* are acceptable arguments to the "+" operator the *Futures* themselves are not. To cope with this problem the Poligon system emulates a machine which has *Futures* as a properly implemented low level data type and all such operations will be coerced so as to wait for the values of the *Futures*. It might be thought, therefore, that this might not buy the user anything, since all *Futures* immediately cause the system to wait for them. Luckily this is not the case. The Poligon system will only wait if the operation which is being applied to the *Futures* is strict on its arguments. Thus functions such as `List` and `Cons` will, happily, create structures such as:

```
List(#<Future 42>, #<Future 127>)
```

i.e. a list which has two elements both of which are *Futures*. This means that the Poligon system will only wait for any value if it is logically necessary. A number of pre-defined functions are defined to be non-strict in some of their arguments. Some of these are given in Appendix E, the appendix concerning data structures.

The cases in which operations are likely to be non-strict are those in which data structures are being created, functions which simply pass their arguments on to other functions or those in which assignments are happening. Clearly the user must be able to define his own functions that are non-strict in their arguments. Poligon provides a mechanism for this.

All Structure declarations made in the Poligon language have a constructor function declared for them which is not strict in its arguments. This constructor function has proper :Keyword arguments, all of which are defined to be non-strict. Thus the following code

```
Structure a_structure Fields field_1, field_2
```

will declare a *Structure* type called *a_structure* with fields *field_1* and *field_2*. A constructor function will be declared called *Make_a_Structure*. It will have the following parameter spec:

```
(&Key Field_1 Field_2)
```

These will be known to the system as being non-strict arguments. This is no use if the user should want to declare his own constructor type functions or such like. For instance if the user wanted to define *My_Cons*, a function which conses backwards then the following definition would not be sufficient

```
Define My_Cons (the_head, the_tail)  
  Documentation "Conses backwards"
```

```

      Cons(the_tail, the_head)
EndDefine

```

This is because, even though Poligon knows that Cons is not strict in its arguments and so no DeFuturing coercion is necessary for the arguments that are passed to Cons, calls to the function My_Cons will still DeFuture their arguments. The way to prevent this is to provide decoration for the arguments to the function. This would be done as follows:

```

Define My_Cons
  (the_head : Non_Strict, the_tail : Non_Strict)
  Documentation "Conses backwards"
  Cons(the_tail, the_head)
EndDefine

```

Arguments are declared to be Non-Strict in exactly the same way as they have types declared for them. Any subsequent calls to *My_Cons* will have the desired behavior.

3.3.2.3. Equality

It should be noted that the presence of *Futures* and the *Multiple-Values* data type in Poligon somewhat changes the possible meaning of equality. Thus EQUAL might well return false even though its arguments represent the same objects. To this end Poligon has its own version of equality and inequality. At the Poligon source code level they are represented by the operators "=" and "≠" (and "%=").¹ These operators are the only safe way to check for equality in Poligon. They apply the minimum amount of DeFuturing necessary.

There are cases when the user might want to have his own definition for equality. Poligon permits this for Structure instances. There are a number of reasons for this:

- A user defined equality predicate for structures might be much more efficient than the system defined test (EqualP).
- The user might have circular structures, which would normally cause stack overflow during equality testing.
- The user might wish to define a criterion for equality which is less strict than equality, for instance coordinates, which are close enough together, might be said to be equal.
- The user might have debugging information in his structures, which are not really part of the simulation. These, presumably, should not be tested for equality.

To do this the user should define a message handler, which supports the *:Equal* message. For an example of this please see the Section 3.3.2.6.

As an optimization the user can define a property called *:Equality-Tester* on the symbol that names the structure type. If this has as its value a function then this is used as the equality testing predicate. If it is the keyword *:No*, then this is an optimized way of saying that there is no special predicate provided.

3.3.2.4. The Size of Structures

It is necessary for the CARE simulator to measure the size of all of the messages that it sends. This includes user defined data structures. Because these can contain elements,

¹If the user chooses to write user functions in Lisp then he should use the functions *Are-The-Same* and *Not-equal*, into which these operators map.

which should not be visible to the simulation, such as debug fields, it is necessary for the user to be able to define his own size measuring mechanism for his structures. This is done by implementing a handler for the *:Total-Word-Size* message. An example of this is shown in Section 3.3.2.6. If there are any fields within the structure, whose size is not constant, or which is not known then this can be determined by calling the function *Total-Word-Size*.

3.3.2.5. The Copying of structures

When the CARE simulator sends a message it copies all of the data in that message. Because user defined structures may have fields, which are not intended to be seen by the simulation, such as debugging fields, it is necessary to be able to define a means of copying user defined structures. This is done by defining a message handler for the *:Copy-Self* message. Within this handler it is the user's responsibility to create a new instance of the structure and copy all of the fields, which are to be copied. An example of such an handler is shown in Section 3.3.2.6. In order to copy the fields, which are to be copied, the user should use the function *Copy-Object*.

3.3.2.6. An Example Message Handler

The following is an example message handler, showing examples of all of the messages mentioned in preceding sections.

- *Coords* are said to be equal if they satisfy the *is-near* predicate.
- The number of words occupied by a *Coord* is 1 + the size of the x and y components.
- When a *Coord* is copied a new one is created but the two instances share the same debugging field.

Coords are printed so that they look like #<Coord: 42, 200>. The x and y fields will be mouse sensitive and they will be printed in a slashified form if the instance in question is being printed in a slashified manner.

```
Structure coord Fields x, y, debug-field

Define coords-are-the-same (a, b)
  is-of-type(b, 'coord) and is-near(a, b)
EndDefine

Define Message-Handler-For-Structure
  (message-name, struct, &Rest, args)
Case message-name Of
  Choice :Equal :
    coords-are-the-same(struct, args•head)
  Choice :Total-Word-Size :
    1 + ;; The name of the structure
    struct•x•Total-Word-Size + struct•y•Total-Word-Size
    ;; Note, not the debug field.
  Choice :Copy-Self :
    Make_coord(:x, struct•x•Copy-Object,
               :y, struct•y•Copy-Object,
               :debug-field, struct•debug-field)
    ;; Note the debug field is shared.
  Choice :Print-Self :
```



```

        format(args•head, "#<Coord: ~a, ~a>",
              struct•x, args•the-third,
              struct•y, args•the-third)
Choice :Which-Operations :
  [:Which-Operations :Equal :Total-Word-Size
   :Copy-Self :Print-Self]
Otherwise :
  ferror(nil, "~S is an illegal operation for ~
           a coord.",
         message-name)

EndCase
EndDefine

Do Putprop('coord, 'Message-Handler-For-Structure,
          'named-structure-invoke)

Do Putprop('coord, 'coords-are-the-same, :Equality-Tester)

```

3.3.2.7. System functions

Along with the change in the meaning of equality a number of frequently used functions become inadequate. For instance, *Assoc* will no longer work in the general case, because not only might the elements in the list or the keys be *Futures* but the list itself might be a *Lazy list*. Thus the user should use the Poligon function *Associate* and other Poligon supplied *Collection* manipulating functions. In particular, the user should **Not** use any functions which process lists internally or which make some sort of test for equality, such as *Member* or *Subst*. The user should use only those functions specified in Appendix E and should write any others himself.

There is a function provided which will do any necessary DeFuturing. This is the only mechanism provided for the user who wants to write his own user functions in Lisp. Although the Poligon system can make sure these functions are not passed any *Futures* as arguments it is quite likely that these arguments will contain *Futures* if they have structure. They may be lists of *Futures*, for instance. In this case the user will have to do his own DeFuturing and accept the risks associated with not writing his code in the Poligon language. The DeFuturing coercion is done by a function called "↑". Thus (↑ something) is guaranteed to have a value which is *neither* a *Future* *nor* a *Multiple-Values* object.

3.3.3. Values

There have been three significant extensions to the language to support new ways of expressing values in expression. These are the mechanisms for creating new *Nodes*, the mechanism for searching the *Blackboard* and *Multiple Values*. There are also values associated with some special identifiers within the user's model.

3.3.3.1. Identifiers and Special Value Denotations

There is only one predeclared identifier in the system. This is, the symbol *Poligon-Blackboard*, whose value is an *Instance* of the class *Root*.

After the declaration of a class the name of that class becomes available to represent the class node of that class. Similarly the name *META-<name>*, where *<name>* is the name of the class, becomes available to represent the *Metaclass* node of the class node.

There is one special value in Poligon. This is the value called *Empty*. It denotes an empty *Value List*. It cannot be used in an expression but it can be used instead of an expression wherever the value of a *Field* is defined. It can therefore be used in a default value specification in a *Class Declaration*, as the right hand side of a *Field* assignment or as the value of a *Field* definition in the *Field Initialisation* component of a *Node Instance Creation* expression.

3.3.3.2. Multiple Values

Poligon has its own representation of *Multiple Values*. This allows it to manipulate multiple values within the CARE simulator and still to be able to deal with system defined functions, which return multiple values.

A *Multiple Values* object is created by the function *Multiple-Values(&Rest values)*. These values can be decomposed by the following means:

- System defined functions (see Appendix N)
- The *MultipleLet* construct, which works for both Poligon *Multiple Values* and Common Lisp multiple values.
- *Definitions* (see Section 3.9.1)

It should be noted that wherever in this manual a function is specified as returning *Values* or *Multiple Values* these are Poligon *Multiple Values*, not Common Lisp ones.

3.3.3.3. The Creation of Nodes

This construct replaces the *Add* change type in AGE and allows a *Node* of an arbitrary type to be created and to have its *Fields* initialized. This can, therefore, be thought of much like a *Cons* or a structure creation operation. The value of such an expression is a new *Instance* of the class of node being created. The construct is introduced by the keywords *New Instance of*. An example of its use can be found in Section 3.6, the section relating to signal data input.

The *New Instance of* construct takes as its first argument the class of *Node* to be created. There are two options for this argument, one which is generalized and one, which is optimized. If the argument has as its value a *Class Node* then the *Class Node* will be used to control the creation of the node. If, however, a large number of instances are being created - this is particularly significant for the input of raw signal data - then an optimized version is available. This requires that the user specify the name of the class to be created, as opposed to the *Class Node*, which is to do the creation (e.g. *'Sheep* as opposed to *Sheep*). This is faster but has some restrictions. This will not be optimized if the initialization for the new *Node* includes any references to any *Fields* in the *Class Node*. What is more, there is now no longer a guarantee that the *Class Node*, which owns this *Node*, will be made aware of its existence before the *Node* starts active processing. Another restriction is that the new *Node* will not be taken from the *Class Node's* resource of *Recycled Nodes*. Thus any user defined resource management will not be of any use.

The creation of *Nodes* is a significant problem in parallel systems because multiple, asynchronous requests to create the same node might appear. These might result in the system having, for instance, a number of *Nodes* that represent what is in fact the same object. It is possible that, had the user's model known about the existence of such an object already it might not have created the new node for it. Poligon provides modifiers to the *New Instance of* construct, which allow the user to perform creations that are regulated and atomic with respect to the *Class Node* doing the creation.

The *New Instance of* construct allows four optional components, introduced by language keywords.

- | | |
|-----------------------------|---|
| Unless | This takes as its argument an expression, which is evaluated on the <i>Class Node</i> doing the creation just before the creation would happen. If this expression evaluates to nil then the node in question is created, otherwise the value of the expression is returned by the <i>New Instance Of</i> construct. |
| Updated Class Fields | This takes as its argument a collection of field updates. These updates are executed on the <i>Class Node</i> after the node has been created. It is not executed if the <i>Unless</i> part has a value other than nil. This component allows the user to cache information on the <i>Class Node</i> concerning which nodes are under its control. For instance, such a cache might contain a mapping table between the numbers painted on the sides of sheep and the nodes that represent them. Within these field updates the new node may be referred to by the name <i>The-Created-Node</i> . |
| Subsystem Of | If the <i>Subsystem Of</i> option is not supplied then the new Node will have the class node, which created it, as its only <i>Supersystem</i> . If the <i>Subsystem Of</i> component is supplied then the values given will be used <i>instead of</i> the class node. Thus if you would like the class Node to be one of the <i>Supersystems</i> then you must specify it in this component as well as the others. |
| Initialisation | If the <i>Initialisation</i> component is supplied then the values of the <i>Fields</i> specified are set. This can, if required, override the <i>Default</i> values. |

The *New Instance Of* construct returns two values. The first is either the new node that was created or the value of the *Unless* component if it was non-nil. The second is a flag, which is *t* if a new node was created and *nil* if no new node was created.

Some examples of the use of the *New Instance of* construct are shown below. For these examples it is assumed that the class *Sheep* confers upon its instances fields called *Colour* and *Number*. Similarly it is assumed that the class *Sheep* has a *Metaclass*, which confers on it the field *Sheep-Number-Map*.

```
New Instance of Sheep
  Subsystem Of : a-flock, my-animals
  Initialisation : Colour ← "Black"
```

This first example causes a new *Sheep* to be created, which is part of *a-flock* and of *my-animals* and which is black. It should be noted that the use of the name *Sheep* in this example is an expression, whose value is a Node of the type class, i.e. it is any *Class Node*. The value supplied here does not have to be a constant "Name" of a class Node, though some optimization can result from this being a constant.

In the second example, below, a sheep is created only if there is no element in the *Sheep-Number-Map* field's latest value, which matches with the number of the new sheep. If there is no match then the node is created and the *Sheep-Number-Map* is updated so that the new node has been cached there. It is assumed that the name *Number-of-the-new-sheep* represents the number painted on the side of the sheep.

```

New Instance of Sheep
  Unless : Associate (Number-of-the-new-sheep,
                    Sheep@Sheep-Number-Map)
Updated Class Fields :
  Sheep-Number-Map ← List (Number-of-the-new-sheep,
                          The-Created-Node)
Subsystem Of : a-flock, my-animals
Initialisation : Colour ← "Black"
                Number ← Number-of-the-new-sheep

```

3.3.3.4. Searching the Blackboard

Poligon supports a mechanism by which the *Nodes* of the *Blackboard* can be searched. This mechanism introduces with it another basic data type in the Poligon system other than the *Node*. This is the *Bag*. *Bags* are described in detail below. It should be noted here that there is a data type similar to *Bag*, called *Set*. *Sets* are very similar to *Bags* except that they contain no duplicate elements. None of the operations mentioned in this section have *Sets* as their values but these data types are freely inter-convertible.

One of the problems in Poligon-like systems is the need to find all of the *Nodes* that satisfy a particular condition and then do something with them. The finding of these *Nodes* is itself a combinatorial problem and so Poligon attempts to provide a high level interface to the parallel mechanisms in the underlying machine, which should allow these searches to be done more efficiently.

There are two methods of achieving these matches; one of which is very general and one of which is optimized both in the linguistic sense and also in the run-time sense in order to achieve simple and frequently made sorts of searches. An example of the more specialized case is as follows

```

Definitions :
  A-Bag ≡ Subset of Sheep For Which Element • Id ≠ nil

```

In this case a *Definition* is being made that specifies that *A-Bag* will represent the *Subset* of the members of the class *Sheep* for which the "•" component of the *Id Field* is not *nil*. That is to say for which the *Latest* value of *The-Sheep's Id field* ≠ *nil*. In this construct the user must provide a collection of things to look at, which can be a class or a *Collection* of *Nodes*. He must also provide a *Field Selection Operator*, which causes the operation to apply to the relevant part of the *Field* value and an operator and value, which will be applied to the value extracted from the *Field*. The operator would normally be a boolean operator but need not be. All elements in the collection represented by, in this case, the expression *Sheep*, which delivered a non-*nil* result from the test specified will be included in the resulting *Bag*. The position occupied by the Identifier *Sheep* may be any expression. If that expression has the value of a *Node* then the *Subset* created is of the *Subsystems* of that *Node*. If the value of the expression is a *Collection* then the *Subset* created will be the *Subset* of the *Nodes* represented by the *Collection*. If the expression represents a list of *Link Cells* then the collection used for the *Subset* operation will be the *Nodes* at the other end of the *Links*.

The above construct returns a *Bag*, each element of which has three values:

- The node which satisfied the predicate supplied.
- The value of the predicate, for instance, in the case of the example above, the value of the expression *Element • Id ≠ nil*.

- The value extracted from the field using the operator supplied. In the example above this would be the *Latest* value of the *Id* field.

An example of the more generalized form of this value generator is as follows:

Definitions :

```
Another-Bag = Subset of Sheep Which Satisfies
              λ(a-sheep)
                a-sheep.Id = The-Black-Sheep.Id
            Endλ
```

In this case *Another-Bag* is the *Subset* of all of the *Subsystems* of the *Sheep*, which satisfy the condition that the *Sheep's Latest Id* is the same as the *Latest Sheep-Id* of *The-Black-Sheep*. *The-Black-Sheep* is a free reference in the predicate and a closure will thus be formed. Another form of this is allowed for the convenience of the user, which allows the negation of the predicate's value, so that the code below and that above are equivalent.

Definitions :

```
Another-Bag = Subset of Sheep Which Fails
              λ(a-sheep)
                a-sheep.Id ≠ The-Black-Sheep.Id
            Endλ
```

The above two constructs return *Bags*, each element of which has three values.

- The node which satisfied the predicate supplied.
- The value of the predicate, for instance in the case of the example above, the value of the call to the λ -expression.
- The a second value returned by the λ -expression if it returns one.

The primary value of a *Subset* construct is a data structure called a *Bag*. A *Bag* can be thought of as a set except that duplicate elements are allowed. *Bags* have a structure which is not of importance to the user but they have a number of operations that define their behavior. *Bags* are intended to allow the exploitation of concurrency by the use of *Futures*. Thus, elements in the *Bag* may not have been determined by the time that the model wants to manipulate them. There are said to be two sorts of elements; *Determined elements* and *Undetermined elements*. The system will always try to return a *Determined element* as the value of an access function.

Undetermined elements can be of two classes, those that will eventually result in a useful object and those that are *Ghost* responses from those elements in a class that did not match a *Subset* operation. There are no *Determined Ghost Elements*.

The fact that *Undetermined elements* can be *Ghost Elements* need not be a problem. *Bags* can still be supplied as arguments to *Subset* operations and *In Parallel For Each* constructs. The major problem with this is that there are a number of possible meanings to the concept of *Number-Of-Elements*. Thus there are a number of functions which deliver different interpretations of this idea. It is conspicuous that calls to *Current-Number-Of-Elements* may well return different values over time.

Other than for the discarding of *Ghost Elements*, *Bags* can be thought of as user-immutable constant data structures. The functions, which manipulate *Bags* are specified in Appendix E, the appendix relating to data structures.

3.3.4. The Structure of Nodes and their Fields

The *Poligon blackboard* is made up of a number of *Nodes*. This section describes the structure of these *Nodes*.

Nodes are, in many ways, like Pascal records. They are data structures, which have fields that the user can access. They have, however, more structure than this. To start with, the fields in the record are of two main types; user defined and system defined. These *Fields* are similar other than that the *System Defined* ones can only be read by the user, since they contain important system data. These are specified in Section 3.4.3, the section relating to System *Field* Names.

The *Fields* of a *Node* have structure, which is not normally visible to the user but can be in certain circumstances. In AGE each *Field* could contain a list of elements, which denoted the history of values associated with that *Field*. This is implemented in Poligon by the value in each *Field* being a list of values. If the user is only interested in one value then he need use only the first element of this list. The behavior of the history list in AGE was well defined, since every element was always implicitly time ordered. This cannot be the case in Poligon, since events can happen asynchronously and can disrupt the order of the list due to computational delays. Poligon provides a rich set of mechanisms for defining much more complex behavior for *Fields*

3.3.4.1. The Basic Behavior of Fields

Poligon provides a basic type of field. This has a *Value List*, but nothing more. The user can then specify three operations on a per-*Field* basis to specialize the *Field's* behavior. These are mentioned below and an example is given in Section 3.4.2.1.

InsertIf	Decides whether a new element should be inserted, when an attempt is made to insert an element. An example of this might be "Only insert the element if it is not <i>Nil</i> ".
RemoveIf	Decides whether an element should be removed, when a removal operation is attempted. An example of this might be "Remove this element as long as its removal won't leave the <i>Field Empty</i> ".
ModifyWith	Determines which elements should be added to the existing elements and which should be removed, given a list of new elements and the existing elements. This is a generalized mechanism, which is provided for when the two mechanisms mentioned above are insufficient. This function must return two values. The first should be a list of the elements to add to the values in the <i>Field</i> , the second should be a list of the elements to remove.

These operations are implemented as functions, which takes three. The first argument should be the element to add/remove or the list of elements to be added as appropriate. The second element is the existing value list and the third is the node itself, so that other fields can be read if necessary. Thus the user could provide the function *Is-Not-Nil* — as follows:

```
Define Is-Not-Nil (something, values, node)
  Ignore(values, node)
  something ≠ nil
EndDefine
```

3.3.4.2. The Sorting Behavior of Fields

Extra functionality can be mixed in with that mentioned above. It is possible to make *Fields* keep their elements sorted. This means that the element returned by the "." operator would be the one which had the highest value of some (user defined) attribute. It would also mean that the "⊕" operator would return the value list in the required order.

The user can enable this behavior by providing either, or both of the following specifier functions.

- SortedBy** This function must be a predicate of two arguments. Its default value is ">".
- KeyedBy** This function, given an element in the value list must be able to extract the value, which is to be passed to the sorting predicate. Its default value is the identity function.

An example is given in Section 3.4.2.1.

3.3.4.3. The Dictionary Behavior of Fields

This mechanism allows *Fields* to act also like a dictionary. This mechanism is enabled by providing the following:

- IndexedBy** A function of one argument, which takes an element which is to be put into the *Field* and computes the index for the field in the dictionary.

An example is given in Section 3.4.2.1.

The user is allowed to index on values, which can legally be tested for *EQL*-ness within the CARE system, namely symbols, numbers and *Remote Addresses*. If the system knows that the indexing function returns an integer and it knows the range of that integer then it can make a much more efficient implementation (using an array instead of a hash table). To do this the user should tell the system about the result type of the function. Thus, for instance, if the indexing function is to be *My-Indexer* and its result range will always be from 5 to 10 then the following will allow the system to optimize this.

```
Do proclaim ([function My-Indexer [t] [integer 5 10]])
```

To gain access to this dictionary mechanism the user uses a modified version of the "." operator. If, for instance, the *Position Field* of a *Sheep* is encoded as a list whose elements are (x, y, time), then its *IndexedBy* function would be *The-Third*. An example of seeing where a sheep is at a given time might be as follows.

```
a-sheep•Position At a-given-time
```

The value of this expression is a *Multiple-Values* object, containing all of the elements in the *Field*, if any, which can be found at *a-given-time*.

3.4. Class Declarations, the Shape of the Blackboard

The *Blackboard* in Poligon is structured in quite a complex manner. This complexity has been kept from the user to the greatest extent possible such that simple applications should only need simple facilities but more complex applications may require of the user a more

sophisticated conceptual model for the shape of the *Blackboard*. To cope with these differing requirements this section has two subsections. The first will describe the *Blackboard* in AGE-like terms and will give simple examples to show how the *Class Declarations* for a simple *Model* might be implemented. The second section will give a more detailed treatment, describing all of the facilities available.

3.4.1. An AGE-like model for the Poligon blackboard

AGE itself has a simple model for its *Blackboard*. The *Blackboard* consists of *Nodes* of different types. These *Nodes* are used to represent the different elements in the solution space of the model. The different types of *Node* are said to belong to *Levels*. The name *Level* is simply used to denote the conceptual level of abstraction to which the *Nodes* belong and the type of those *Nodes*. The *Blackboard* is imagined as being split horizontally into these levels of abstraction. The rules in the system then act on the *Nodes* on the *Blackboard*.

In Poligon the equivalent of the *Levels* of AGE are called *Classes*. This name is intended to indicate that the *Blackboard* can be split vertically into different types of *Nodes* at the same level of abstraction. A class can be thought of very much like an AGE level. A class declaration is a template for the *Instances* of that class, showing the structure of a *Node* of that class. An example might be as follows

```
Class Definitions For Model "My Farm-Yard Model" :  
  Class Sheep :  
    Fields : Number-of-legs : 4  
             Weight  
             Colour
```

The collection of class declarations in a model are introduced by the keywords *Class Definitions For Model*. In this case only one class is being defined for the model called "My Farm-Yard Model". Thus all of the *Nodes* being dealt with on the *Blackboard* defined above are of the type *Sheep*. An *Instance* of *Sheep* will have three attributes - *Fields* or *Slots*. These *Fields* are used to denote the colour, weight and number of legs of each sheep *Instance*. In this case the declaration is specifying that the initial weight and colour of each new sheep *Instance* is undefined but that all sheep have, by default, four legs. This value can be changed if the model should discover that any given sheep *Instance* has a different number of legs.

There is now a slight difference here from AGE. In AGE the member *Instances* of a *Level* were kept in a list, which was not part of the *Blackboard*. In Poligon, however *Classes* are represented as first-class citizens. For more information on the full implications of this statement you should refer to Section 3.4.2. This means that, in fact, the class itself is represented as a node. This node has the same name as the name of the class. Thus the symbol *Sheep* in this example denotes the node, which is "in charge of" the *Instances* of the class *Sheep*. This node has *Fields* just as the *Instances* have. By default it has only the system defined set of *Fields* (see Section 3.4.3 on *System Field Names*) but it can also have user defined *Fields*.

The fact that the class of *Sheep* in the system is represented as a *Node* is only important once the model starts to want to manipulate all of the sheep together. For example the expression

Sheep@Instances

Has as its value a list, which contains all of the *Instances* of the class *sheep*.

3.4.2. A Detailed Description of the Class Structure

To have a full understanding of the *Class* structure of *Poligon* it will first be necessary to understand the following definitions.

Superclass/Subclass This is used in its conventional object-oriented sense.

All objects, which is to say *Nodes*, in *Poligon*, are members of *Classes*. This class is much like a "type" in languages such as Pascal in as much as it defines both the shape of the data structure and the operations that may be performed on it.

Classes can be specialized. This means that if one defines a class of *Farmyard-Animals* such that all animals in a farmyard are of this type a specialization of this type might be *Sheep*. In this case *Sheep* would be said to be a *Subclass* of *Farmyard-Animals* and, by symmetry, *Farmyard-Animals* is said to be the *Superclass* of *Sheep*. In practice any class of objects can have any number of *Superclasses*. *Sheep* might have *Edible-Animals* as a *Superclass* as well as *Farmyard-Animals*. This means that *Instances* of the class *Sheep* would have the characteristics, i.e. the *Fields*, of both *Edible-Animals* and *Farmyard-Animals* as well as any that might be particular to *Sheep*.

Abstract Superclass An *Abstract Superclass* is a class which has no direct *Instances*. It is used only as a *Superclass* of other *Classes*.

Instance/Instance-Of An *Instance* of a class is a data structure, which has the shape defined in a template, called the class. For example, a *Node*, which we shall call "John's Sheep", might be an *Instance* of the class *Sheep*. This means that it is a *Node* in the *Poligon* system, which has all of the *Fields* defined for *Sheep*. "John's Sheep" is said to be an *Instance-of* the class *Sheep* and the class *Sheep* is said to have the *Node* "John's Sheep" as one of its *Instances*. This should not be mistaken for the *Superclass/Subclass* relationship. The *Instance-Of* relationship can be thought of as an "Is a kind of" relationship, whereas the *Superclass* relationship can be thought of as "Is a subtype of".

Supersystem/Subsystem The *Superclass/Subclass* and *Instance/Instance-Of* relationships do not express anything about "Part/Whole" relationships. For example "John's Sheep" may be an *Instance* of the class *Sheep* but it is not *Part* of the class *Sheep*. The class is something, which merely describes the shape of *Sheep* as a whole. It would be reasonable, however, to want to express the fact that "John's Sheep" is part of a particular *Flock*. This is done with the *Supersystem/Subsystem* relationship in *Poligon*. Thus the fact that "John's Sheep" is subservient to "John's Flock" is expressed by the statement that "John's Sheep" is a *Subsystem* of "John's Flock" or that "John's Sheep" has "John's Flock" as one of its *Supersystems*.

Metaclass

Classes are first-class citizens in *Poligon*. They are represented, like everything else, as *Nodes*. These *Nodes* them-

selves are able to act just like any other *Nodes* in the system and so must be *Instances* of *Classes* themselves. The class of which any given class Node is an *Instance* is called, by convention, the *Metaclass* of that class. It is by means of *Metaclasses* that characteristics that are particular to the class itself, as opposed to the *Instances* of that class, are defined.

In Poligon it is possible to declare a very general class hierarchy. Any class can have any number of *Superclasses* or *Metaclasses*. Similarly any *Instance* can have any number of *Supersystems*. The following is a list of the axioms of the class structure.

- All *Nodes* are *Instances* of a class.
- *Classes* are represented as *Nodes*.
- Class *Nodes* are *Instances* of their own *Metaclasses*.
- An *Instance* can only be an *Instance* of one class.
- The *Metaclass* of which a class Node is an *Instance* is created by the system and has as its *Superclasses* all of the *Metaclasses* that the user defined for the original class.
- Only one *Instance* of a class Node is created initially. That Node can be referred to by the name of the class within a Poligon model.
- The Node representing the *Metaclass* of a class called <name> can be referred to in a Poligon model by the name *META-<name>*.
- Any number of *Instances* of any class can be created, including *Metaclasses*.
- *Metaclasses* are *Instances* of the class *Metaclass*.

Some examples will now be shown to help to explain the implications of the above. A simple example has already been shown in the preceding section, showing how to define the class *Sheep*. Here, examples will be given to show how it is possible to define the class *Sheep* in terms of *Farmyard-Animals* and *Edible-Animals*.

```

Class Definitions For Model "My Farm-Yard Model" :
  Class Farmyard-Animals :
    Fields : Weight
           Colour
  Class Mammals :
    Fields : Number-of-legs : 4
  Class Edible-Animals :
    Fields : Price-per-pound
           Suitable-wines
  Class Sheep :
    Superclasses : Farmyard-Animals, Edible-Animals,
                  Mammals
    Fields : Thickness-of-wool

```

Here, *Sheep* are shown to be types of *Farmyard-Animals*, *Mammals* and *Edible-Animals*. Because they are edible they inherit the *Field* which will be used to hold the price-per-pound that they would fetch at the butcher and some wines that might be suitable to accompany their consumption. All farmyard mammals have four legs by default. Poultry, of course, would have only two. *Sheep*, because they are wooly, farmyard mammals have the attribute *Thickness-of-wool*, which is peculiar to sheep.

The node referred to by the name *Sheep* belongs to the class *Meta-Sheep*. Whenever a class is defined by the model the system also defines a *Metaclass* called *META-<name>*, where <name> is the name of the class which has just been defined. This Node can be thought of as the producer of the node *Sheep*. The *Metaclass* to which it belongs can have user de-

defined attributes. This is done by specifying *Metaclasses* for the class. For example, if the class *Sheep* is also to have the characteristics of *Animal-Producers* then one might do the following.

```

Class Definitions For Model "My Farm-Yard Model" :
  Class Farmyard-Animals :
    Fields : Weight
            Colour
  Class Mammals :
    Fields : Number-of-legs : 4
  Class Edible-Animals :
    Fields : Price-per-pound
            Suitable-wines
  Class Animal-Producers :
    Fields : Rate-of-production
  Class Sheep :
    Metaclasses : Animal-Producers
    Superclasses : Farmyard-Animals, Edible-Animals,
                  Mammals
    Fields : Thickness-of-wool

```

In this case any *Instance* of the class *Meta-Sheep* will have as its *Superclass* the class *Animal-Producers* i.e. it is an *Animal-Producer* as well as just a *Meta-Sheep*. Any number of distinct *Instances* of this class can be made. This could be done either because the different *Instances* might have distinct characteristics or because a lot of *Instances* of the class they represent are to be made and in a *Parallel* environment it might be more efficient to have a number of class *Nodes* servicing the requests to create *Nodes*.

3.4.2.1. Field Behavior Modifiers

The field behavior modifiers mentioned in Section 3.3.4 are defined in the following manner in the class declarations.

```

Class Flock :
  Fields :
    Size : 0
    ;;; A Field containing the number of sheep
    ;;; in the flock
    InsertIf : 'Is-Not-Nil
    RemoveIf : 'Is-Not-Nil
    Position : Empty
    ;;; Contains the position of the flock as
    ;;; a list (x, y, time)
    ModifyWith : 'Position-Modifier
    SortedBy : '>
    KeyedBy : 'The-Third
    IndexedBy : 'The-Third

```

Here two *Fields* are defined, one has *InsertIf* and *RemoveIf* specifiers and is a simple *Field*, the second has a modifier and is a sorted *Field* with dictionary behavior.

3.4.2.2. The Subsystem Structure of the Blackboard

As has been mentioned above, the *Blackboard* is structured in terms of *Instances* of *Classes*. There is another structure to the *Blackboard*, which is expressed as a function of *Subsystems/Supersystems* relationships. The *Blackboard* has at the top of the *Subsystems* hierarchy an *Instance* of the class *Root*. It is this Node which controls the *Blackboard* during its *Initialisation*. All of the class and *Metaclass* Nodes are *Subsystems* of this Node, which has the globally visible name *Poligon-Blackboard*. This should not be mistaken for the Node *Root*, which is the class of which the top Node of the *Blackboard* is an *Instance*.

The class *Root* can be user defined. If the user elects not to defined any shape for it then it will be defined suitably by the system but if the user wants to add *Fields* to the *Poligon-Blackboard* Node then he need simply declare a class called *Root*. The system will process this definition correctly so that it will take on the behavior defined by the system as well as that defined by the user.

3.4.2.3. The Fields within Nodes

The names of the *Fields* within a Node are represented by identifiers. These *Fields* can have default initial values associated with them. The definition of a *Default* value is expressed in the following manner.

field-name : <initexpression>

The <initexpression> must be either:

- A comma separated list of expressions that makes no reference to any *Nodes* or their *Fields*, in which case it will be used to define the initial values for the *Value List* of the *Field*.
- The identifier *Empty*. In this case the *Field* is initialized with an empty *Value List*.

The *Initialisation* of *Fields* is completely optional. Any attempt to read an uninitialized *Field* will, of course, result in an error.

3.4.3. System Defined Field Names

Each Node, irrespective of its class possesses a number of fields that are used by the system. Some of these are user accessible and some are not. For obvious reasons only those that can be seen by the user are mentioned here. These fields are read-only. An error will be flagged at compile-time if an attempt is made to update one of these fields.

Clock	The <i>Clock Field</i> always has the current time in it in units defined by the user's input data. This <i>Field</i> is updated each time the clock ticks. This means that rules associated with it will be triggered by time changes.
Subsystems	The value of <i>Subsystems</i> is a list of the <i>Nodes</i> for which the Node in question is a <i>Supersystem</i> .
Number-of-subsystems	The <i>Latest</i> value of Number-of-subsystems is an integer denoting the length of the list of <i>Subsystems</i> .
Supersystems	The value of the <i>Supersystems Field</i> are always the <i>Nodes</i> , which have the Node in question as one of their <i>Subsystems</i> . This, in some way, denotes that the <i>Supersystems</i> are in some way in control of the Node in question or that the Node is a "Part Of" its <i>Supersystem</i> . A

class Node will always have the *Poligon-Blackboard Node* as a member of the value of its *Supersystems Field*.

Number-of-supersystems The *Latest* value of *Number-of-supersystems* is an integer denoting the length of the list of *Supersystems*.

Instance-Of The *Latest* value of this *Field* is always the class Node of which the Node in question is an *Instance*.

Name The *Latest* value of this *Field* is always a string denoting the name of the Node.

The following *Fields* are defined only on *Nodes*, which represent *Classes*.

Instances The values of this *Field* are the *Instances* of that particular class Node. It should be noted that this does not necessarily represent a list of all of the *Nodes* of the shape denoted by the *Class Declaration* for that class, since there may be any number of *Instances* of that type of class Node (*Metaclass*).

Number-of-instances The *Latest* value of this *Field* is an integer denoting the length of the list of *Instances*.

3.4.4. The Printed Representation of Nodes

All *Nodes*, by default, are printed simply with their names and with "#<>"s if they are printed in a slashified fashion. The user may define the printing behavior of instances of a class, so as to display the values in significant fields. This is done by an extension to the *Class Declaration*. An example of this is shown below.

```
Class Definitions For Model "My Farm-Yard Model" :
  Class Sheep :
    Fields : Number-of-legs : 4
            Weight
            Colour
    Display As : The-Node•Name,
               The-Node•Number-of-legs
```

A list of things to be printed out may be provided. Any references to the node in question should be made through the special identifier *The-Node*. These items are printed out, by default, using the "γ" format directive. Thus an instance of the class defined above might be printed out as "#<Sheep-42 4>". If more sophisticated format control is required the user may provide a format string as the first element in the list of items. If this is done the format string will be used instead of the default format directive. An example of this is shown below.

```
Class Definitions For Model "My Farm-Yard Model" :
  Class Sheep :
    Fields : Number-of-legs : 4
            Weight
            Colour
    Display As : "~A has ~A legs", The-Node•Name,
               The-Node•Number-of-legs
```

This will result in *Sheep-42* being printed out as #<Sheep-42 has 4 legs>.

It should be noted that in writing this printing behavior it is often useful to use the Coerce-To-Object or Identity functions so as to remove any references to Multiple Values of Futures that might confuse the printed representation.

3.4.5. The Input Handler Class

Input is handled by *Instances* of an *Input Handler Class*. The *Input Handler Class* is a user defined class like any other and it can act like any other but, when the system initializes, the user designates a class which is to be responsible for the handling of input. All of the input handling is automatic from then on. Input is channelled through the *Input Handler Class* Node, which farms it out to its *Instances*, which it has created to handle the input. It creates as many as it needs in order to be able to deal with the input as it comes in. *Instances* of the *Input Handler Class* are the only ones that are created by the system without the intervention of the user's model other than the *Class Nodes* and the *Poligon-Blackboard* Node. For this reason it may be of use to the user to define the *Input Handler Class* to be a distinct class from any of the other *Classes* associated with the user's model, though this is by no means mandatory. An example definition of an *Input Handler Class* is shown below.

```
Class Input-Handler :
  Metaclasses : Input-Handler-Class-Mixin
  Superclasses : Input-Handler-Mixin
```

This class has no user defined *Fields*. There need be no more code defined by the user than this in order to specify the handling of input except if he should choose to call the class something other than *Input-Handler*. For more information regarding this point please see Section 3.7, the section which specifies the requirements for *User Defined Initialisation*. In order for an *Input-Handler* class declaration to be valid it *must* specify the *Superclasses* and *Metaclasses* shown in the example above.

One reason for attaching rules to the *Input Handler Class* Node or to the *Input Handler Nodes* might be to propagate caches to the *Input Handlers*, which the *Input Handlers* could use to initialize the instances that they create. This could save the original source of the cache from becoming a hot-spot.

3.5. Node Field Selection

The *\$Value* construct provided in AGE does not exist in Poligon. Access to a *Field* is made simply by applying the name of the *Field* to the *Instance*. Thus "an-instance•a-field" will deliver the *Latest* value in that *Field*.

3.5.1. System Defined Field Selection Operators

A new infix function application operator (*Field Selection Operator*) has been defined. This is called \oplus and acts just like the \bullet operator in all cases other than the application of the name of a *Field*. Thus *a-list* \oplus *First* will deliver the *Head* of *a-list*. In the case of the function being a *Field* name the code generated will have the effect on the *Fields' Value List* shown in Table 3—6.

Operator	Effect
\bullet	Extracts the Latest value.
\oplus	Extracts All of the values.

Table 3—6 Poligon Language Field Selection Operators

Clearly \bullet and \oplus are somewhat equivalent to the *Latest* and *All* option in AGE's $\$Value$ respectively and the operations that they denote will be referred to by these names.

It should be noted that these *Field Selection Operators* are special. They do not act like function application, being simply an abstraction mechanism. Thus it is not the case that using prefix function application notation will achieve any of these effects.

As is always the case in Poligon the *Field Selection Operators* have synonyms, which are easy to print out on printers that do not support Sail characters. These are shown in Table 3—7.

Normal	Easily Printed
\bullet	@
\oplus	@@

Table 3—7 Poligon Language Field Selection Operators and their easily printed representations

3.5.2. System Defined Field Selection Predicates

A number of operators have been implemented using the field selection operator mechanism but which it is not anticipated that the user will use as normal postfix function application operators, though they would work as such. These are a set of predicates used to test certain characteristics of the values of fields.

- Is-Empty** This predicate returns *T* if the value field being examined is *Empty* otherwise *Nil* is returned. If the field is uninitialized then an error will be reported.
- Is-Not-Empty** This predicate returns *Nil* if the field being examined is *Empty* otherwise *T* is returned. If the field is uninitialized then an error will be reported.
- Is-Undefined** This predicate returns *T* if the field in question has not been initialized, otherwise it returns *Nil*.
- Is-Not-Undefined** This predicate returns *Nil* if the field in question has not been initialized, otherwise it returns *T*.
- Is-Empty-or-Undefined** This predicate returns *T* if the field in question has not been initialized or it has the value *Empty*, otherwise it returns *Nil*.
- Is-Not-Empty-or-Undefined** This predicate returns *Nil* if the field in question has not been initialized or it has the value *Empty*, otherwise it returns *T*.

3.5.3. User Defined Field Selection Operators

In order to support the change to the semantics of postfix function application operators an extension to the L100 base has been provided in order to allow the user to define his own operators of this kind. If the user wanted to define an operator, called β , which would extract the second time ordered element, as opposed to the most recent or all of the elements, then it would be done as follows:

```
Declare-Field-Selection-Operator('β , :Second, :Poligon)0
```

Where the above is the L100 code to perform this process, β is the name of the operator, *:Second* is a *:Keyword* denoting the name of a method, which is defined below and *:Poligon* denotes the name of the language in which this operator is to be defined. The

above procedure call must be made before the *Compiler* sees any instance of the use of the operator. This can be forced using the *Execute* construct.

```

Poligon:
(Define-method (basic-slot :Second)
  (&Optional (index nil))
  "Returns the second element from the value list."
  (if index
    (ferror nil
      "This sort of slot does not understand ~
      indexing."
    )
    nil
  )
  (if (equal Values :Undefined)
    (ferror nil
      "Cannot find an element in the ~
      uninitialized slot ~A."
      (send self :Name)
    )
    (if (or (not Values) (not (rest Values)))
      (ferror nil "Cannot find an element in ~
        the empty slot ~A."
        (send self :Name)
      )
      (Second Values)
    )
  )
)
)

```

Such methods must be defined for the flavor *Basic-Slot* and must take one argument, which is the index used by the dictionary behavior. Any non-nil value for this argument should cause an error. It is generally not necessary to have to resort to declaring such operators.

3.5.4. Reading more than one Field at a Time

Because of the fine grain of critical sections in Poligon there may be problems in getting consistent values from remote nodes over time. Thus a Poligon program which has an expression, such as:

```
a-function(a-node•a-field, a-node•a-field)
```

cannot guarantee that the values passed to *a-function* will be the same, since two distinct reads will be made and *a-node* might have been updated between these two read operations. This immediate problem can be solved by the caching of the value of *a-node•a-field* by the use of a *Definition*. However if *a-function* needed the values from a number of fields of *a-node* then clearly the caching solution cannot work. For this reason Poligon provides a mechanism for reading the values of any number of fields atomically, so that a consistent set of values can be derived.

```
a-node & •field-1 & •field-2
```


The above expression will return *Multiple Values* of the *Latest* values of the *field-1* and *field-2* fields. For more information on *Multiple Values* please see Section 3.3.3.2 and Appendix N.

3.6. Data Input

The *Data Input* component of the model is that part of the Poligon system that receives signal data. All signal data goes through this process, though the process itself could be instantiated any number of times in a *Parallel* environment. The process handles all input invisibly to the user. The Poligon system, however cannot know how to put the data, which has been read, onto the *Blackboard* and to this end it calls a pair of user defined routines to do the work for it. The first of these is called *Time-Of-Input-Record*. It takes a single argument; the record read in from the input. It must be able to extract the timestamp from the signal record and return it to the system. The system will then do all of the relevant operations to scale this time relative to the system's real-time clock and compensate for any slippage caused by stopping the system for debugging sessions. An example of this function follows

```
Define Time-of-Input-Record (Record)
  Record•Head
EndDefine
```

In this case the signal record is a list and the timestamp held in the first element of that list.

The second user-defined component of the *Data Input* component is a procedure called *Input-Procedure*, which knows how to turn the signal record, which has been read, into a suitable representation on the *Blackboard*. The procedure is passed the signal record, the timestamp of that signal record and the *Input Handler* Node, which is controlling the input, as arguments. This procedure is expected to be able to decode the signal record so that it can determine what sort of signal record has been passed to it and then put the data onto the *Blackboard*. This is done by creating a Node of a suitable class and initializing its *Fields*. The *Input Handler* node argument is supplied so as to support system dependant initialization of the new nodes, for instance with information about the caches being kept on *Class Nodes*. An example of this procedure follows

```
Define Input-Procedure
  (Record, Timestamp, The-Input-Handler)
  Ignore(The-Input-Handler)()
  Case Record•The-Second Of
  Choice :Sheep :
    New Instance of Sheep
    Initialisation :
      Serial-Number ← Record•The-Third
      Colour        ← Record•The-Fourth
      Position      ← list(Record•The-Fifth,
                          Record•The-Sixth,
                          Timestamp)
    Otherwise : Ferror(nil, "Illegal input record type.")
  EndCase
EndDefine
```

In this case, the signal record is represented again as a list. The second element of the list is a record type selector, which if it is a *Sheep* causes the system to create a new *Instance* of a *Sheep* Node with the relevant *Fields* initialized. If the record is not for a *Sheep* then the

system flags an error. This procedure must be written in the Poligon language, since Lisp procedures do not have access to the *Instance* creating constructs.

It should be noted that, by default, the *Initialisation* of these values will cause events on all of the initialized fields. This may not be desired and so the optional pragma *NoEvent* can be used selectively to disable the automatic event generating mechanism. Thus:

```
Time ← NoEvent Record•Head
```

Would not cause an event on the *Time Field*.

There is no constraint as to when these routines are defined in the source file, other than that they must be after the *Class Declarations* (see Section 3.4) and they must be loaded by the time the system is run. The rest of the model should never use them.¹

3.7. User Defined Initialisation

Most of the *Initialisation* for the Poligon system is done automatically. There is, however a need to provide a hook for user defined *Initialisation* code. The user must specify an *Initialisation* code body, which can, of course, be null.

```
Initialisation :  
  Debug-Format ("~&The system is starting up") ◊  
  do-my-initialization ()
```

In this case the user has used the mandatory Poligon language *Initialisation* construct to define a code body, which prints out a message and calls a procedure whilst the system is initializing itself.

The system will only start up properly if the user has declared an *Input Handler Class* called *Input-Handler*, since the system will assume that the name of the class of input handlers is *Input-Handler*. If you decide to call it something else you will have to specify this change in the *Initialisation* code body by the use of the following form. For more information on the *Input Handler Class* please see Section 3.4.5.

```
Set-Input-Handler-Class ('My-Input-Handler)
```

In this case the user has called the class of *Input Handlers* "My-Input-Handler". For more information regarding the *Input Handler Class* please see the documentation associated with class declarations.

When the system starts it tries to get the name of an input file. It does this by looking for a function called *Get-Input-File-Name*, which takes a single argument, which is the name of the system specified in the *Class Definitions*. If this function is defined then it is called, otherwise the user is prompted for a file name.

Get-Input-File-Name can be defined by the user and it is expected to deliver either a string or pathname denoting the file to be used for signal data, or nil in which case the run is aborted. An example definition of this function is given below. It prompts the user with a menu with the names of two test files and keeps doing so until the user selects one of the options.

¹Note. It may well be useful to the user to timestamp his data when it enters the system with the time of the input record.

```

(Defun Poligon-User:Get-Input-File-Name (system-name)
  (if (equalp system-name "My System")
      (w:menu-choose
        ' ("First"
          :value "My-System:My-System;First.Input"
          :documentation "The First Test"
        )
        ("Second"
          :value "My-System:My-System;Second.Input"
          :documentation "The Second Test"
        )
      )
      :Label " Which Test ? ")
  )
  nil
)
)

```

After this has been selected, the system starts up.

3.8. User Defined Abort Function

There is no termination condition in the Poligon system, as there is in AGE. This is because the system will continue indefinitely, unless it terminates because of a lack of any more simulation events. There is, however, a hook provided by the system so that user defined code can be executed when the system is aborted (terminates itself). The system can be aborted even after it has terminated of its own accord. This hook allows the user to clean up after the model has stopped. This could be used particularly for the closing of trace files, and such like, that the user has opened during the simulation. All that the user need do is define a function called *User-Abort-Function*, which should be a function of no arguments.

3.9. User Defined Knowledge

The definition of knowledge is done in *Knowledge Source* constructs. A *Knowledge Source* is really only a way of parcelling up rules. The *Knowledge Source* itself has no effect on the behavior of the system but is intended to give the user a higher level handle on his rules. A *Knowledge Source*, consists of two main parts; a *Definitions Part* and a collection of Rules. Before the structure of *Knowledge Sources* is considered in any detail mention should be made here of Poligon's *Definitions* mechanism.

3.9.1. Definitions, Lazy and Eager and Forced Evaluation

In AGE the rule writer tended to make "Bindings" within the condition part of a rule, which were carried through into the right hand side. There was no way of specifying a binding without the code that defined its value being executed. For a value that was expensive to calculate this was most wasteful if the rule did not fire, since the value might not have been used. This caused the rule writer to hack the problem so that the definition was made by side effects in the middle of conditions, and such like, in the action parts of rules. This is obscure and hacky. The Poligon system does not allow this sort of conduct, but provides the rule writer with a more powerful construct instead. This is the lazily evaluating *Definitions* mechanism. At a number of points in the model the user is allowed to make *Definitions*. These *Definitions* associate values with names. The values can be derived by any expression. Thus

Definitions :
a-name \equiv 3 + 4

associates the name "a-name" with the code that, when executed, will deliver the value of the expression 3 + 4. Wherever a *Definitions Part* is legal, any number of *Definitions* can be made. It is important to note that this is not a binding in the conventional sense. It is not possible to assign to a-name later in the code of a rule. What is more, the code associated with the calculation 3 + 4 is only executed when the value of *a-name* is first needed. If it is never needed in the course of the execution of the rule then this code will not be executed. Any further references to *a-name* will get the value derived by the first evaluation without it being recomputed. Thus the user gets the best of both worlds; he gets the clarity associated with defining names to have immutable values and he gets the optimization associated with doing work only if it is absolutely necessary.

A further example might be as follows

Definitions :
field-a, field-b \equiv
a-node & •node-field-1 & •node-field-2

In this case the names *field-a* and *field-b* name the values in the fields *node-field-1* and *node-field-2* in the node *a-node*. Putting more than one name to the left of the equivalence sign causes a *Multiple Values* object to be unparcelled to give up its values. This allows *Definitions* to denote the values of fields, which are read atomically in the manner shown above. It should be noted, however, that in the following expression the name *a-name* does not denote the value of the *node-field-1* field, it denotes the *Multiple Values* object, which contains the values of both of these fields.

Definitions :
a-name \equiv a-node & •node-field-1 & •node-field-2

The strategy of not evaluating expressions unless they are needed is referred to as *Lazy Evaluation*.

Lazy Evaluation in the Poligon system is provided in three forms; there are *Definitions*, *Lazy Function Arguments* and *Lazy lists*. *Lazy Lists* are user definable lazily evaluated list structures. These are defined fully in the appendices.

In a parallel, real-time environment, in which side effects are, sadly, being made all over the place asynchronously, there can be problems associated with the meaning of lazily evaluated quantities, which would not exist in a purely functional, non-real-time environment. Even in functionally programmed systems with full *Lazy Evaluation* it is usually considered necessary to provide a *Force* operator, which, when met by the system, causes its argument to be evaluated fully. This can be of quite some significance in functional programming engines, be they real or virtual, since large amounts of space can be consumed by holding on to lots of partially reduced Lambda expressions, which represent values that may eventually be needed but which are not needed yet.

In a purely functional environment the presence of operators, such as *Force*, have no effect upon the semantics of the program. They are added as decoration to functional programs simply as tuning features. Poligon provides two facilities akin to a force operator. These are the *Eager* and the *Force* operations. It should be noted, however, that the use of these facilities can have an effect on the semantics of a program.

The reason for the use of the *Force* facility is to ensure that a particular definition represents the value that you want at the Time that you want it, since the value represented by that expression might change over time and the unevaluated components of an expression can be passed around indefinitely by the use of the *Expectation* mechanism.

Eager Evaluation is that method of evaluation, which causes evaluation of a quantity to be started before the value is needed. This can happen even if the value is never accessed at all. As is the case in any system, which supports eager evaluation, this opens the door to doing a lot of unnecessary work. What is worse, there is one form of side effect which is allowed in *Definitions*. This is the creation of new *Nodes* on the *Blackboard*. *Eager* evaluation can be induced by the *Eager* construct, which will not wait for the results of the eagerly evaluated *Definitions* or the *Force* operation, which will wait until all referenced expressions are completely satisfied.

It should be noted that the use of the *Eager* or *Force* constructs on a definition, which refers to a Node creation expression could result in the generation of unwanted *Nodes* if the code that refers to the definition is never passed.

The *Forcing* or *Eager* evaluation of *Definitions* is only allowed in a few places in the Polygon language. Unlike a fully lazy language, there are a limited number of things which can be *Forced* meaningfully. These are the definition items. The user is allowed to perform *Force* or *Eager* operations in the places where the semantics of such a *Force* or *Eager* are well defined. Since there is an implicit serialization associated with the execution of a *Force* or *Eager* operation *Forcing* or *Eager* evaluation is allowed only in those places where such a serialization might be meaningful. Thus a *Force* or *Eager* construct can occur immediately before the *When Part* of a rule is tested, immediately before the *If Part* of a rule is tested, immediately after the *Action Part*, *Otherwise Part* or *Timeout Part* of a rule is entered and immediately after the body of an *In Parallel For Each* component is entered. Any defined item may be *Forced* or *Eagerly* evaluated, but only after it has been defined. Thus, within the body of an *In Parallel For Each* construct *Definitions* made at the head of the *In Parallel For Each* can be *Forced* or *Eagerly* evaluated, as can those made in the *Action Part* of the rule, in the *Rule Header* and at the *Knowledge Source* level.

An example of a *Forcing*, in the *Action Part* of a rule might be as follows

```
Action Part :  
  Definitions :  
    Four = 2 + 2  
  Eager : a-definition  
  Force : Four, another-definition
```

In this case, when the *Action Part* of this rule is entered, the *Definition* for *a-definition* will start to be evaluated but the execution of the *Action Part* will not wait for its result to be returned. *Four* and *another-definition* will then be *Forced*, that is, evaluated and execution will wait until they are fully satisfied. It is assumed that *another-definition* and *a-definition* have been defined either at the rule level or at the *Knowledge Source* level, above.

3.9.2. The Definition Part of a Knowledge Source

The definition part of a *Knowledge Source* is not quite like the definition part of any other construct in the Polygon language. This is because the *Knowledge Sources* themselves are compile-time rather than run-time entities. The *Definitions* that are made at the *Knowledge Source* level are visible within all of the rules in the *Knowledge Source* but they are, like the *Definitions* in rules themselves, evaluated in the context of the rule's execution and not

in the context of the *Knowledge Source*. Thus the rule writer need not fear that the evaluation of a definition within one rule will be visible and already evaluated within another rule. For instance if the following *Knowledge Source* level definition were to be made:

```
Definitions :
  a-value  $\equiv$  <a> + <b>
```

where <a> and are expressions, a reference to *a-value* within one rule invocation in that *Knowledge Source* might not get the same value as a reference in another rule even another invocation of the same rule. The values of *a-value* are, of course, constant during the invocation of each rule.

3.9.3. The Declaration of Rules in a Knowledge Source

An example of a rule and a *Knowledge Source* might be as follows

```
Knowledge Source An-Example-Knowledge-Source
Definitions :
  The-Serial-Number, The-Flock  $\equiv$ 
    The-Sheep & •Serial-Number •?Flock

Rule : Tell-the-user-when-a-sheep-has-moved
  Class      : Sheep
  Field      : Position
  Condition Part :
    When      : The-Position  $\neq$  :Unknown and The-Flock  $\neq$  nil
    If        : The-Serial-Number  $\neq$  nil
  Action Part :
    Changes   :
      Change Type      : Update
      Updated Node     : The-Flock
      Updated Fields   :
        Sheep-Positions  $\leftarrow$  List(The-Sheep, The-Position)

  Otherwise Part :
    Execute :
      Polygon-Format
        ("An unnumbered sheep ~A has moved.~%",
         The-Sheep)
```

Here there is one definition at the *Knowledge Source* level that is used inside the rule. There are two forms of rule; normal rules, which will be described here, and *Expectation* Rules, which are largely the same but will be described in Section 3.9.3.10, the section concerning *Expectations*, in order to put them into their correct context.

Rules have four distinct parts; a *Rule Header*, a *Condition Part* and an *Action Part* and an optional *Otherwise Part*. The *Rule Header* specifies the class of the entity to which the rule is associated, a *Field* to which it is attached and allows *Definitions* to be made, which are local to the rule. The field specified can only be a *Field* of *Instances* of the class specified, either directly or through *Superclasses* of that class. The rule will be attached to all *Instances* of the specified class.

3.9.3.1. Special Names

Within the scope of a *Knowledge Source* one special name is available. This is *The-Triggering-Node*. Within the context of a rule it always represents the Node which caused the *Event* on the *Field* which triggered the rule.

Within the scope of the "*Updated Class Fields*" part of a "*New Instance Of*" construct the identifier *The-Created-Node* represents the node, which has either been created by the operation or selected as a result of the *Unless* part.

Within the body of a rule, after the specification of the Class and the *Field* to which the rule is to be associated, two special names are available to the user. These are derived from the names of the *Field* and class specified. They are derived by prefixing *The-* to the names given in the class and *Field* specifications. They represent the Node that experienced the *Event* and the values of that *Field* that changed when the *Event* was caused. In the case of the above example the name *The-Sheep* is an example of the *THE-<class-name>* construct and refers to the *Sheep* Node on which the rule hangs, to which an *Event* happened. If the value of the *Value List* in the *Field*, which had the *Event* caused to it, is *Empty* then the value of the *THE-field-name* identifier will be the *:Keyword :Empty*.

In all of the contexts in which names mentioned above are legal the user may, instead use the names *The-Node* to denote the node for which the rule in question is being triggered and *The-Field* or *The-Slot* to denote the *Latest* value in the field which was triggered.

3.9.3.2. The Rule Header

The header of a "Normal" rule consists of three components, one of which is optional and two of which are mandatory. The format of the headers for *Expectation* rules will not be covered here, since they are described fully in Section 3.9.3.10.

The first and second components are mandatory, they are specifications for the class and *Field* with which the rule is to be associated. The rule will be associated with each Node on the *Blackboard*, which is an *Instance* of the class specified.

The third element in the *Rule Header* is the *Definitions Part*. It is optional and acts just like the *Definitions Part* for *Knowledge Sources* only the *Definitions* made are only visible within the defined rule.

3.9.3.3. The Condition Part

The *Condition Part* of the rule consists of two mandatory parts and four optional parts. These are the *When Part*, the *If Part* and their associated *Forcings* (see Section 3.9.1) and the *Select Part* and its associated *Forcing*. These represent three separate parts of the pre-condition. The *When Part* is executed as soon as an *Event* happens on the *Field* to which the rule is attached. It is intended to refer only to the Node that has been affected. Thus it can be evaluated quickly. If this clause delivers a value other than *nil* then the firing of the *If Part* is scheduled. It is by no means guaranteed that the *If Part* will fire "immediately" after the *When Part*. The *If Part* may refer to any Node in the system, though doing so may result in communication delays. If the *If Part* of the rule delivers a value other than *nil* then the rule will schedule the *Action Part* to fire if no *Select Part* has been defined. If a *Select Part* has been provided then this is evaluated. Again there is no guarantee that the *Action Part* will fire immediately after these. If the value of the *If Part* predicate is *nil* or the *Select Part* expression, if it has been provided, does not select an *Action Part* body then the *Otherwise Part* of the rule is scheduled for firing, if the rule has an *Otherwise Part*.

The *Select Part*, if it is provided, is used to select one of a collection of *Action Part* clauses. This is explained more fully in the next section.

For information concerning *Forcing* components please see Section 3.9.1.

3.9.3.4. The Action Part

The *Action Part* consists of three main components, a *Definitions Part*, a part in which actions take place as a direct consequence of the *Condition Part* of the rule being non-*nil* and, optionally, a part which is invoked if the *If Part* of the rule evaluates to *nil*.

In the example above the *Action Part* of the rule has no *Definitions* part. Such a *Definitions Part* would be present to allow *Definitions* that are specific to the *Action Part* in a place that seems meaningful. Of course the *Lazy Evaluation* mechanism guarantees that there would be no difference in semantics if an *Action Part* level definition was moved to the rule level.

The *Action Part* of the rule can have two distinct formats. The first occurs if the user does not provide a *Select Part* in the *Condition Part* of the rule. In this case the user can define any number of *Changes* or *Execute* components. These *Changes* or *Execute* components will be run in *Parallel*. No serial dependencies should be allowed between them. A similar collection of *Changes* and *Execute* components can be defined in the *Otherwise Part* of the rule. These will be executed if the *If Part* of the rule fails or the *Select Part*, if provided, does not find a matching *Action Part* body.

If the rule provides a *Select Part* then the value of the expression in the *Select Part* is used to match against keys specified before any number of *Action Part* bodies, like those described in the preceding paragraph. The system will execute the first *Action Part* body whose key = the value of the *Select Part*, checking the keys in the same order as the lexical order as in the source program. At most one such body will be executed, such bodies are, therefore, mutually exclusive in their operation. An example of a rule with a *Select Part* and multiple *Action Part* bodies is shown below.

```
Knowledge Source An-Example-Knowledge-Source
Definitions :
    The-Serial-Number, The-Colour =
        The-Sheep & •Serial-Number & •Colour

Rule : Tell-the-user-when-a-sheep-has-moved
    Class      : Sheep
    Field      : Position
    Condition Part :
        When      : The-Position ≠ :Unknown
        If        : The-Serial-Number ≠ nil
        Select    : If The-Colour = :Black
                    Then :A-Black-Sheep-Has-Moved
                    Else :A-Non-Black-Sheep-Has-Moved
                    EndIf

    Action Part :
        :A-Black-Sheep-Has-Moved :
            Execute :
                Poligon-Format("Black sheep ~A has moved.~%",
                               The-Serial-Number)
        :A-Non-Black-Sheep-Has-Moved :
```



```
Execute :
    Polygon-Format ("Sheep ~A has moved.~%",
                    The-Serial-Number)
```

```
Otherwise Part :
    Execute :
        Polygon-Format ("An unnumbered sheep ~A has moved.~%"
                        The-Sheep)
```

In this example, a message is printed out to show that a *Sheep* has moved as long as the *sheep's* position is known. If the *sheep* has no known *serial-number* then a suitable message is printed out. *Black sheep* are given special treatment, as one would expect. Within a *Changes* component the user may specify any number of *Change* components. These are defined to operate in series. Each *Change* represents an update to some *Node* or *Collection of Nodes*, which can entail a modification to any number of its *Fields* or an *Event*, which is to happen to a *Node*.

One of the types of *Change* is the *In Parallel for Each* construct. This does not change just one *Node* but allows the user to define a change to be made to a collection of *Nodes*. This should not be viewed as a loop, since its components are executed in parallel. No serial dependencies should be allowed between the elements. Within such a construct a local symbol denotes the value of the element of the collection of *Nodes* being considered.

Within the body of a *Change* five forms of *Change Type* are permitted. These are as follows:

Cause Events	Allows the user to cause an event on the field, that is, to make it look as if the <i>Field</i> has changed, without changing the value in the <i>Field</i> . This is a form of semaphore to the rules that are attached to that <i>Field</i> .
Discard	This allows <i>Nodes</i> , which the user is no longer interested in to be (somewhat) switched off and made invisible to <i>Subset</i> operations.
Expect	This is covered in Section 3.9.3.10, the section concerning <i>Expectations</i> . It is a mechanism for the allocation and specialization of knowledge dynamically.
Recycle	This allows <i>Nodes</i> , which the user is certain have been finished with, to be reused.
Update	Allows the user to update any of the <i>Fields</i> in the referenced <i>Node</i> .

3.9.3.5. The Cause Events type of Change

The *Cause Events* type of change is very simple. *Events* can be caused on any number of *Fields* which can be seen by the *Node* under consideration. An example of such a change might be as follows

```
Changes :
    Change Type      : Cause Events
    Updated Node     : an-expression-delivering-a-node
    Updated Fields   :
        A-Field
        Another-Field
```

In this case two *Fields* in the Node defined by *an-expression-delivering-a-node* are being told that *Events* have happened to them. The *Fields* are referred to by name. In this case they are *A-Field* and *Another-Field*. The value of *THE-<Field>* when a rule is triggered by such an event is a multiple values object whose only value is the Node, which caused the event.

3.9.3.6. The Discard Type of Change

This change type allows *Nodes*, which have been finished with, to be discarded. This operation is similar to, but should not be confused with, the *Recycle* type of change, mentioned in Section 3.9.3.7. This operation does not in any way delete the Node in question. It does however have the following effects:

- It deletes any links coming out of the Node in question.
- It has the Node removed from the *Subsystems Field* of its *Supersystems* Node.
- It has the Node removed from the *Instances Field* of its class Node.
- It disables the clock on the node, so that clock driven *Expectations* and pending timeouts will not be activated.
- It unhooks the Node from all of the rules, which are associated with it.

This means that, although the model is still allowed to read from and write to the Node, rules, which watch these fields will not be activated.

This *Change Type* should be used with caution. An example of its use follows.

```
Change Type   : Discard
Updated Node  : a-node
```

In this case the Node *a-node* is discarded.

3.9.3.7. The Recycle Type of Change

This change type allows *Nodes*, which have been finished with, to be recycled. It is very difficult to determine in any general sense whether a Node can be recycled so Polygon opts for a user defined mechanism. Even in this case, it is difficult to determine whether a Node has been finished with and can safely be recycled. The only case in which the user can be sure that it is safe to recycle a Node is when the rules associated with the class of Node in question are all mutually exclusive in their operation, no other rules in the system read from or write to this class of Node and all of the activities in all of the rules are complete. Thus this mechanism is usually only meaningful at the lowest level of abstraction in the model at which signal data enters the *Blackboard* and propagates its data upwards and has rules which only fire once on the Node, when it is created. Even in this case it is only safe to recycle the Node if the *Action Part* of the rule has only serialized change types, or if the parallelized change types read *Fields* of the Node in question through *Forced Definitions*.

An example of the use of this construct follows

```
Change Type   : Recycle
Updated Node  : a-node
```

In this case the Node *a-node* is recycled. When a Node is recycled it is reset, such that all of its *Fields* are reinitialized. Any links coming out of this Node are removed (see Appendix R for more information concerning links). It is then removed from the *Subsystems Field* of its *Supersystems Nodes* and from the *Instance-Of Field* of its class

node kept for the time when the next call to the *New Instance of* construct for that class of Node is made, when the recycled Node is used instead of creating a new node.

3.9.3.8. The Update Type of Change

In an *Update Change* the user defines those *Fields* that are updated. This is by far the most commonly used *Change Type*. This is done by specifying a collection of fieldname/*Update Operator*/value triples. An example of this might be:

```
length ← 3 + an-expression, another-expression
```

This causes the system to attempt to add the values derived from the expression $3 + \text{an-expression}$ and the value of *an-expression* to the values of the *Field* called *length*. This is akin to the *Modify* operation in AGE. One of the major differences in Poligon, however is that the user is not constrained to one type of update as he was in AGE. The nature of the update performed, as a result of these attempts is, by default, simply adding them to the *Value List*. More complex and conditional modifications can be defined by the user on a per-*Field* basis. This is explained in Sections 3.3.4 and 3.4.2.1.

It should be noted that the default mechanism in Poligon is to cause an event, that is to say to enable the triggering of any associated rules, on any *Field* that is successfully updated. It is possible that, for some reason, the user might want to update a *Field* without triggering the rules that are watching it. To do this the user should use the optional pragma *NoEvent*. If the above example was explicitly expected not to cause an event then it would appear as follows.

```
length ← NoEvent 3 + an-expression, another-expression
```

It should be noted that this facility should be used with caution, since it might compromise the modularity of the system. This facility is, in a sense, related to the *Cause Events Change Type*.

Because the user might want to specify a number of arguments to an update operation which are determined at run-time Poligon provides a mechanism to support this. This is somewhat equivalent to the *Apply* function in Lisp. This is achieved by the use of the "&" modifier. The following two lines of Poligon code are equivalent.

```
length ← 3 + an-expression, another-expression  
length ← & List(3 + an-expression, another-expression)
```

There are a number of predefined *Update Operators*. These represent the operations associated with the addition of elements, the removal of elements and the replacement of all of the elements in the *Value List*.

The representations of these operators and their default effects on an old value *x* with operands *y* and *z* are shown in Table 3—8. It should be noted that this behavior can be modified significantly by user defined behavior for *Fields*, thus this table represents only the default mechanism. For more information on how to modify this, please see Sections 3.3.4 and 3.4.2.1.

Operator	Name	Effect
$x \leftarrow y, z$	Modify	$List(y, z) \leftrightarrow x \rightarrow x$
$x \leftarrow^* y, z$	Remove Element	$(x \text{ Without } y) \text{ Without } z \rightarrow x$
$x \leftarrow\text{---} y, z$	Replace Value List	$y \rightarrow x$ [only 1 operand allowed]

Table 3—8 Polygon Update Operators and their effects

To illustrate these operators in operation a few examples are given in Table 3—9. In these examples it is assumed that the field, which is being operated on is called *x*. The values in the *Value list* of this field will be shown in braces, such as "a b c".

Values in x	Operation	New values in x
{a b c}	$x \leftarrow d$	{d a b c}
{a b c}	$x \leftarrow d, e$	{d e a b c}
{a b c}	$x \leftarrow^* d$	{a b c}
{a b c b}	$x \leftarrow^* b$	{a c}
{a b c}	$x \leftarrow\text{---} List(d, e)$	{d e}

Table 3—9 Examples of the use and effects of Update Operators

These *Update Operators* have readily printable synonyms associated with them. These are shown in Table 3—10.

Normal	Easily Printed
\leftarrow	<-
$\leftarrow\text{---}$	<---
\leftarrow^*	<-*

Table 3—10 Polygon Language Update Operators and their easily printed representations

It should be noted that the operation " $\leftarrow\text{---} list(d, e)$ " is only a shorthand for " \leftarrow Empty" followed by " $\leftarrow d, e$ " or " $\leftarrow \& list(d, e)$ ".

An example of the user definition of an *Update Operators* is specified below:

```
Declare-Update-Operator('← , :Modify, :Polygon)
```

The first argument given to this function is any legal operator specification. The second is a keyword denoting the name of a method for the flavor *Basic-Slot*, which knows about how to make the update to the *Field*. The *:Keyword :Polygon* simply denotes the language for which this operator is being defined.¹

3.9.3.9. The "In Parallel For Each" Construct

The *In Parallel For Each* construct is the closest that the Polygon language gets to a "loop". This construct is not strictly a *Change*, it is a way of expressing a number of *Changes* of the same type that are to be done to a number of *Nodes*. As is the case with most "loop" constructs the *In Parallel For Each* construct uses a declared identifier to represent the ele-

¹Note: The user is strongly discouraged from writing his own *Update Operators*. The user should be able to get all of the desired behavior by writing his own behavior for his fields. For information on this issue, please see Sections 3.3.3 and 3.4.2.1.

ment from the original collection that is being considered. Thus will have the effect of executing the *Update Change* to each element in *a-flock-of-sheep*.

```
In Parallel For Each lamb In a-flock-of-sheep
  Change Type : Update
  . . . . .
```

Within the scope of the *Change* the identifier *lamb* represents the only element of the original list which is visible to the *Change* as an individual. It should be noted that the name of this construct was chosen to stress that it should not be thought of as a loop.

The expression which delivers the collection of *Nodes* to be considered, in this case *a-flock-of-sheep*, can deliver any collection or Node. If a Node is delivered then it denotes the *Subsystems* of that Node.

3.9.3.10. Expectations

It is difficult to define what is meant by *Expectations* in the AGE sense within the metaphor of Poligon. What is generally meant to be the case with an *Expectation* is that the user is supposed to be able to watch a specific piece of information and wait for it to change, such that when it changes some sort of action takes place. It is a form of attention focussing mechanism.

In Poligon this is interpreted as being a requirement to watch particular fields on particular *Nodes* and wait for them to change. Posting an *Expectation* in Poligon, therefore, means assigning a rule at run-time to a particular Node and attaching it to a particular *Field*. A special syntax has been defined to allow this as a *Change Type* and rules themselves are allowed to take *Arguments* in order to support the transfer of context as might be required by the user. This is described below.

The *Expectation* mechanism allows the user's model to parcel up as much context as he wants at the time that the *Expectation* is posted. What the user has to do is to specify the name of the rule that is to be attached, the Node to which it is to be attached and the *Field* that it is to monitor. After this there are a number of optional items that the user can provide in order to customize the rule that represents the *Expectation*. An example of an *Expectation* being posted follows.

```
Changes      :
  Change Type : Expect
    Rule      : Watch-To-See-If-This-Is-A-Flock-Leader
    Node      : The-Sheep•Flock
    Field     : Position
    When      : The-Expected-Node•Sheep-Positions =
                The-Sheep
    If        : t
    Active    : t
    Delete    : nil
    Timeout   : The-Flock•Clock + 10
  Definitions :
    The-Current-Position = The-Sheep•Position
```

In this case an *Expectation* is being posted within a rule, which is associated with the class *Sheep*. The rest of this rule is not shown but the identifier *The-Sheep* refers to the Node for which the rule is being fired. A rule called *Watch-To-See-If-This-Is-A-Flock-Leader* is

being attached to the *Position Field* of the Node designated by the expression *The-Sheep•Flock*. The idea, here, is to watch the flock associated with the sheep in question in order to see whether its position tracks that of the sheep.

Two preconditions are provided. The *When Part* is an expression which is executed before the rule attempts to fire its own *When Part*. The *When Part* of the rule will only be tried if the *When Part* defined above evaluates to non-*nil*. The constraints on the *When Part* defined here are the same as those on the *When Parts* of rules described in Section 3.9.3.3. The same sort of behavior applies to the *If Part*. It is determined before the *If Part* of the target rule is attempted. The *If Part* and then *When Parts* of *Expectation* expressions can be any expressions. They may refer to the Node to which they have been attached by the identifier *The-Expected-Node* and they may refer to the value of the *Field* to which the rule has been attached by the name *The-Expected-Field*. Either of these items may be omitted, in which case they will be assumed to be "t".

There are four more optional components. These are discussed below.

Active	This is a predicate which determines whether the <i>Expectation</i> is active or not. It is not allowed to see the names <i>The-Expected-Node</i> or <i>The-Expected-Field</i> . It can, on the other hand see an identifier called <i>The-Time</i> . This has the value of the time at which the <i>Expectation</i> is trying to fire itself. If the <i>Expectation</i> is supposed to become active only after a certain time then this expression provides a means to denote this. If this section is defaulted then the value "t" will be assumed. This means that the <i>Expectation</i> will always be active.
Delete	This is an expression denoting a predicate which, when non- <i>nil</i> will cause the association between the rule and the Node to be removed. This component is evaluated after the <i>When Part</i> of the rule has been evaluated. This means that if the value of this expression is non- <i>nil</i> - the default is <i>t</i> - then the association between the rule and the Node will be broken after the first attempted firing. In the case above the rule will never be disassociated with the Node in question. The default case is akin to a "Try-Once-Only" sort of firing. It should be noted that the value of this expression does not in any way affect the firing of the rule to which it is attached. It only serves to prevent <i>further</i> rule firings by dissociating the rule from the Node, which is being watched.
Timeout	The value provided here should be an expression, whose value is a number. If the value is not a number then this will be interpreted as meaning that the <i>Expectation</i> is to have no <i>Timeout</i> . The number, which is the value of the <i>Timeout</i> component, represents an absolute time after which the <i>Expectation</i> is to time out. In this case the <i>Expectation</i> will time out after ten time units from now. When the <i>Expectation</i> times out it is detached from the Node which it monitors and the <i>Timeout Part</i> of the <i>Action Part</i> of the rule is executed if it is present. The time specified is the absolute time in the same units as the time in the user's input signal data, for more information on signal data see Section 3.6.
Definitions	These are <i>Definitions</i> just like those anywhere else in a rule, in that they associate names with values. These allow the

user to define *Arguments* to rules and to pass context into those rules.

As was mentioned above, rules are allowed to take *Arguments*. These are only meaningful in the context of *Expectations*. The *Rule Header* for a rule, which is intended to be used as an *Expectation* rule is different from that for a normal rule. In the case of an *Expectation*, the rule is not associated with any particular class, Node or *Field*. Where normal rules have defined, within their bounds, identifiers which denote the Node in question and the value of the *Field* in question, so do *Expectation* rules but they are referred to by the generic names *The-Expected-Node* and *The-Expected-Field*. An example *Rule Header* for an *Expectation* rule is shown below.

```
Rule : Watch-To-See-If-This-Is-A-Flock-Leader
      Arguments : The-Current-Position
```

In this case the rule, which was referred to above is being defined. It has two *Arguments*. Note that these *Arguments* have the same names as those used above in the *Definitions*. The *Arguments* are associated with their actual values by name and not by position. Within the rules the *Arguments* act just like items defined in a *Definitions Part*. This means that an *Argument* to a rule is not necessary unless the evaluation path of the rule requires its value. If an *Argument* has not been specified for a value, which is needed, an error will be flagged at run-time.

The *Action Part* of an *Expectation* can have an extra construct, which is a *Timeout Part*. This is executed if the *Expectation* times out. An example *Action Part* for this rule might be as follows:

```
Action Part :
  Execute :
    Debug-Format("~&The expectation has fired.")
Timeout Part :
  Execute :
    Debug-Format("~&The expectation timed out.")
```

3.10. Atomic Operations and Critical Sections

Poligon is a system which goes to considerable pains to minimize the length of critical sections, so as to enhance parallelism. However, there are times when the user may need to know that certain operations are performed atomically so as to guarantee consistency or meaningful behavior. This section enumerates the operations provided by the system, which are guaranteed to be atomic, and also mentions how, to some extent, user defined code can be executed within a critical section.

There are three main types of atomic operation:

Field Selection The execution of any field selection operation is atomic. This applies both to the reading of single fields and multiple fields. Arbitrary user defined, side-effect free code can be executed within a user defined field selection operator. A form of atomic read/write is provided as an extension to field selection operations. This allows atomic "test and set" operations to be implemented. An example of this form is as follows:

```
a-node•a-field :
```

Unless : a-node.a-field = 42

Updated Fields :

a-field ← 42

The semantics of this is as follows. All reads to the node in question and the values of those reads are cached. If an *Unless* component is provided then this is evaluated and, if its value is *Nil*, the update is performed. If no *Unless* part is provided the update happens unconditionally. Once the update has been performed the original values read from the fields are returned.

Thus, in the example above, the field *a-field* is set to 42 if it wasn't before and the original value is returned.

By means of this mechanism user defined locks can be implemented. No reference to nodes other than the node in question are permitted in the right hand side of the update component, other than by means of definitions. All definitions specified in the right hand side of the assignment are evaluated *before* entering the critical section.

Field Updates

The execution of an *Update Operator* is atomic. It should be noted, however, that this does not guarantee the atomicity of the evaluation of the arguments to that operator. For instance, in the following update component all of the right hand sides of the operators are evaluated first, and then *all* of the updates are performed together, atomically.

Updated Node : a-node

Updated Fields :

field-1 ← <expression-1>

field-2 ← <expression-2>

This has the effect that, although all of the updates happen atomically the contents of the fields being updated may change between the times that the expressions are evaluated and the update is made. For this reason it is not advisable to write code of the following form.

Updated Node : a-node

Updated Fields :

field-1 ←---

if a-value Is-In a-node@field-1

then a-node@field-1

else a-value ←

a-node@field-1

endif

To achieve this sort of thing the user should either provide *Field* behavior modification functions as specified in Section 3.4.2.1 or, if the user is really desperate, an *Update Operator* of his own.

Node Creation

One of the things which it is very important that the system should be able to do atomically is the conditional creation of nodes. A full exposition of the syntax for this process can be seen in Section 3.3.3.3 so this section will dwell on the exact evaluation model for the optional Initialisation, Unless and Update components.

The Unless and Updated Class Fields components of a *New Instance Of* are both evaluated in the same way. Any definitions present in these expression are evaluated in the con-

text of the current execution, outside the critical section in which the node creation occurs. The remainder of these expressions is evaluated by the *Class Node* within a critical section. No references to any nodes, other than the class node may, therefore, be made in these expressions, other than through the definitions, which will already have been evaluated. As usual for update operations, the right hand sides of the update components are evaluated before the updates happen, though, unlike a normal update, the right hand sides are evaluated atomically in the same critical section as that in which the *Update Operators* are executed.

The Initialisation component is evaluated in a slightly different manner from the Unless and Updated Class Fields components of a *New Instance Of*, since the initialization must happen to the node which has been created, not to the *Class Node*, which has done the creation. As is the case with the Updated Class Fields and Unless components, all definitions referenced within the body of the initialization expression are evaluated in the context of the current execution, before the critical section is entered on the *Class Node*. When the critical section is entered on the *Class Node* all subexpressions which are field reads and refer to the same expression that is being used to denote the class node in the overall *New Instance Of* expression are evaluated on the *Class Node*. This means that these reads will be executed atomically before any of the updates are made. Once this has happened the new node is created in such a way that the remaining right hand side expressions for the initialization is executed atomically on the newly created node.

4. The Poligon Run-time System and how to use it

The Poligon system consists of two components; the Poligon *Compiler* and the *Run-Time System*. The language defined by the former is described in Section 3. The latter is described here.

The *Run-Time System* for Poligon has two main parts. First there is the underlying architecture that supports the Poligon system in either its *Serial* or *Parallel* modes. This provides the *Blackboard* itself, handles signal input and initializes the system. Secondly there is the user interface. This is a constraint frame with a number of panes, which have a number of mouse sensitive features, which are intended to allow easy debugging and control of the system. There is a trace and breakpoint facility, which is integrated into the user interface. The system comes with what, it is hoped, is a rich set of facilities to help the debugging of *Models*.

4.1. How to get a Model to run under Poligon

When the Poligon system is loaded, both the *Serial* and *Parallel* mode run-time systems are loaded along with the compiler. This is done as follows.

```
(Make-System 'Poligon :Noconfirm :Silent :Nowarn)
```

The user is advised to use only the *Serial* mode until his model is well debugged. The *Parallel* version is hugely slower. It is also less easy to debug Poligon *Models* in the *Parallel* environment.

Once the Poligon system has been loaded the user's model can be loaded. The model has to be compiled with the Poligon *Compiler*. The generated code must be compiled using the Common Lisp compiler. During rapid development it may be easiest to do this by doing a *Meta-X Compile Region/Buffer* on the compiled model when it is loaded in *Zmacs*. If it is quite stable, however, it may be worth compiling as a file. The generated code from the user's model is generally rather large; it may take some time to compile. The model should compile without errors, though if any *Knowledge Source* level *Definitions* are made that are not used within a given rule then a free reference warning will be given, though this is not fatal. Once it has been compiled the model must be loaded. This is done automatically if it is compile by using *Meta-X Compile Buffer*. This leaves the system in a state in which it is ready to be activated. The Poligon *Control Frame* will have been created, though it will not be activated and is not selectable until the system is started up.

To start up the system you should execute the function:

```
(Run-Poligon)
```

to get the *Serial* mode or

```
(Run-Poligon t)
```

to get the *Parallel* mode of operation. The user can always chop and change between the *Serial* and *Parallel* modes by executing this function in the appropriate form.

The Poligon system will then be activated, the *Control Frame* will be activated and exposed and the system will stay quiescent until the user selects an option such as *Run* (see Section 4.3.3 on the Command Menu).

4.1.1. Resetting Poligon

At times the Poligon system may get confused, either due to the window system locking up or due to the *Knowledge Base* getting confused by incremental changes made during debugging. It is therefore useful, at times, to reset parts of the Poligon system in order to circumvent some of these problems. This is done using the procedure *Reset-Poligon*. A call to this procedure will ask a number of questions of the user, which ask which parts of the system should be reset. The user should not attempt to run the Poligon system after a call to this procedure without using the *Run-Poligon* procedure to start it off. If the Poligon *Control Frame* has been recreated then it will not be selectable until the *Run-Poligon* procedure has been called. Clearly if the *Knowledge Base* is reset the user's knowledge will have to be reloaded.

4.2. The Underlying Architecture

The underlying architecture of the Poligon system has been mentioned in the introduction to this document. Here a little more detail will be added and the differences between the *Serial* and the *Parallel* modes of operation will be given.

The *Nodes*, be they *Class Nodes*, the *Poligon-Blackboard* Node or normal *Nodes*, are implemented as instances of *Flavors*.

No *Nodes* are defined at load time. The *Poligon-Blackboard* Node, which is the first Node to be created by the system, when it initializes a simulation run, is an *Instance* of the class *Root* and has special knowledge about the shape of the *Blackboard* because it has access to a data structure which holds information concerning the identities of all of the *Classes* and all of the rules associated with them.

When the *Poligon-Blackboard* is created and the system is initialized it creates *Instances* of *Class Nodes* to represent each of the user defined *Class Nodes* and any system defined *Metaclasses*. The rules for the system are then distributed so that each class knows about the rules that are to be associated with *Instances*. Once the system has initialized itself an *Event* is signalled on all of the *Subsystems* and *Instances Fields* of the *Nodes* that have been created. This allows user defined *Initialisation* rules, which are associated with these *Fields* to fire at this point.

Once this has happened the user's *Initialisation* code is executed, the signal data file is opened and the system starts up properly.

Rules are implemented as instances of rule flavors.

Each time a *Rule's* invocation gets past the *When Part* a context packet is created for that specific invocation. This is a specialized Node-like Poligon object of the flavor *Context*.

All actions within the system are performed by message passing, though this is not visible at the language level. Even the reading of a *Field* value on the same Node causes the system to send a message to itself, though this special case has been optimized.

4.3. The Poligon User Interface

The Poligon User Interface is intended to provide an easy to use method of observing the operation of the user's model and debugging it. Poligon constructs a constraint frame, the *Poligon Control Frame*, in which to display all of this information. Once the Poligon frame has been exposed it can be reexposed by typing "<System><escape>". The constraint frame consists of a number of panes, which are described in this section.

4.3.1. The Lisp pane

This is the large pane on the right of the frame and is used for the display of trace information and user debug output. It is a Lisp Listener window, so the user can type any necessary system commands in this pane. If tracing is enabled for any component of the system then it is displayed in this window. The user can write into this pane by means of a number of printing procedures, which are specified in Appendix O. This pane has *More* processing disabled by default.

4.3.2. The CARE Pane

This is a pane which contains any CARE instruments that are being used by the system. It can be found below the *Lisp pane*. If the CARE system has not been loaded and, therefore, Poligon is being run in its *Serial* mode then this pane is not configured in. For information on the meaning of the CARE instruments the user is advised to consult the relevant CARE documentation.

4.3.3. The Command Menu

The *Command Menu* is on the bottom of the frame on the left. It is by the use of this menu that the user selects those facilities that are concerned with the overall control of the system. Each item has documentation in the "who-line" but a complete list of all of the implemented commands with a brief description is given below.

Abort	This causes the execution of the user's model to be aborted.
Circuit	This command pops up a menu and allows the user to select a Circuit (Design) to use for the simulation. This command is only meaningful in the Parallel mode.
Configuration	This command pops up a menu and allows the user to select a configuration for the Poligon.control frame. A number of configurations are available; some with CARE instrumentation and others without. Poligon will typically pick a reasonable configuration for its display, so this command is used mainly as an override.
Dribble Off	This command causes the <i>Dribble File</i> , specified by the <i>Dribble File</i> option of the <i>Parameters</i> command to be closed so that dribbling stops. (see <i>Dribble On</i>)
Dribble On	This command causes all output to the <i>Lisp pane</i> , the <i>Message Pane</i> and to *Trace-Output* to be sent also to a file named by the <i>Dribble File</i> option of the <i>Parameters</i> menu. If a dribble file is already open then this is closed and a new one is opened.
Log	This command will cause a file to be written which has a printout of the current state of the <i>Blackboard</i> . The file written is that named by the <i>Log File</i> option of the <i>Parameters</i> command. The clock is not stopped during the

execution of this command. It is intended to be used once a breakpoint has been found.

Parallel

This command sets Poligon to run in *Parallel* mode. It is equivalent to the procedure call (*Run-Poligon t*).

Parameters

This command allows the user to select the values for a number of CARE and Poligon system simulation parameters and a number of variable attributes used in the system, such as file names for printing things out. The action of these is listed below.

Display All Data Structures A number of Poligon internal data structures contain detail which is not of importance to the debugging of *Models* as a whole and which cause the screen to be cluttered. If this flag is set then all of the structure of these internal data structures will be displayed. If it is reset then the important information that they contain will be displayed but their structure will be displayed minimally.

Allow All Trace Within Abortable Code If this flag is set to *Ignore* then whenever Poligon enters any code which might be aborted trace operations are disabled. If it is set to *Only when successful* then the printing of the trace messages generated in abortable code is only allowed after the section has been exited without being aborted. This saves confusing trace messages from being printed out. The default value for this flag is *Only when successful*.

Enable Breakpoints Within Abortable Code If this flag is set then whenever Poligon enters a breakpoint within any code which might be aborted the breakpoint will take effect. Normally the user will not want such breakpoints to happen but in hunting for obscure bugs he might. The default value for this flag is *No*.

Trap On Errors Within Abortable Code If this flag is set then whenever Poligon enters an abortable code segment and finds an error it will trap. This is not normally a desirable state of affairs, since whilst experimenting Poligon might well try to execute code with very bogus arguments. Occasionally it might be useful to the user, however, since erroneous code might be called only within abortable segments. The default value for this flag is *No*.

Trace If Segment Takes Longer Than n usecs If this is an integer then a message will be printed out if a segment of user code took longer than *n* microseconds to execute.

Inspect Details Of Structures If this flag is set then any data structures whose print methods use mouse-sensitive printing facilities will be printed in a fully mouse-sensitive manner in suitable windows. These include inspector and debugger windows.

- Action on Deadlock** This controls what happens when a deadlock condition is found in the *Serial* mode. Three options are provided: Stopping with an error, notifying the user in the trace window or ignoring the condition.
- Metering** Selecting this option will cause the execution of the model to be metered when running. For information on the metering system please read the system documentation for the metering package.
- Metering Micro Enables** This option allows the user to set the value for the %meter-micro-enables used when metering. For more information on this subject the user is directed to the relevant TI documentation.
- Count Down Space in Metering Partition** It is typically the case that the size of the metering partition supplied in any given machine will only be large enough to allow the metering of a few seconds of Poligon application code execution. Since having metering enabled slows execution down considerably it is often a good idea to enable this option so that the user can see when the metering partition is filling up and can abort the run. Once the metering partition is full, no more useful data will be gathered.
- Start Metering After Data Time** Because it is generally only possible to meter small amounts of application code, it is usually a good idea only to meter a point in the simulation's run when it is known that something worthy of metering is happening. If this parameter is set to a number then metering will not be enabled until after the domain time specified by that number. Metering will then stay on continuously until the metering partition is filled.
- Compilation Factor** This is a number that the user can specify which will allow the user to experiment with the system whilst pretending that the compiler is better than it really is. A value of 2, for instance, would indicate that the system should pretend that user evaluations take half as long as they really do.
- Parallel Metering Slow-Down Factor** When running with metering enabled the CARE simulator's idea of timing is skewed by the slower execution of the user's code. This parameter allows the user to compensate for this by telling the system that, when metering is enabled, it is to scale its timings by the factor specified.
- Jump the clock when idle** This option is only meaningful in the *Serial* mode. When this option is set to *Yes* the clock is advanced to the time at which the clock would next tick if there are no outstanding events to be processed in a given clock tick period.

Internal Debug Messages Enabled	If this flag is set then any Polygon internal debug print messages will be printed. This feature is used primarily for system development purposes or the finding of Polygon system bugs.
CARE Debug History	If this flag is set to the <i>Enabled</i> value then the CARE simulator's internal debug history recording mechanism is enabled. The default value for this parameter is <i>Disabled</i> . This option is primarily of interest for Polygon system debugging and is unlikely to be of much use in the debugging of Polygon models.
CARE Evaluator Processing	This sets the value of the CARE internal flag which determines the time taken to evaluate a body of code in a CARE evaluator. A value of <i>Nil</i> , the default, for this parameter will cause the actual code's execution to be timed.
Automatic GCing	This flag is used in order to control <i>Garbage Collection</i> . It can take three values; <i>When Necessary</i> , in which case the system will execute a <i>GC-Immediately</i> when it is determined that it needs more memory [<i>Parallel</i> mode only]; <i>Temporal</i> , in which case the normal, temporal GC is used, or <i>Never</i> , in which case the GC is switched off altogether. The latter can be useful for timing tests of one sort or another or for runs, which are known to be short enough that the system will not run out of memory, but a long, slow <i>GC-Immediately</i> might be executed because the GC monitor does not know that the simulation is nearing its end.
Window Fonts	This is a list of the fonts in which the user would like different sorts of output to be displayed. The list is in the following order: Trace information, General system information, User output.
File Fonts	This is a list of the fonts in which the user would like different sorts of output to be printed to files. The list is in the following order: Trace information, General system information, User output, CARE messages.
Log File	This is the name of the file used by the <i>Log</i> command.
Input File Default	This is the name of the file used for a default for the signal input file if the user has not supplied the function <i>Get-Input-File-Name</i> .
Statistics File	This is the name of the file used by the <i>Statistics</i> command.
Dribble File	This is the name of the file used by the <i>Dribble On</i> command.
Serial Clock Rate	This is the number of system 60Hz clock ticks which are equivalent to one tick of user input data time, when running in the <i>Serial</i> mode.

- Parallel Clock Rate** This is the number of CARE system clock ticks which are equivalent to one tick of user input data time, when running in the *Parallel* mode.
- Future Force Rate** This value is used only in the *Serial* mode. When the *Scheduler* decides that it is appropriate, *Futures* are evaluated. When this happens a block of unsatisfied *Futures* are evaluated. This number indicates the number of unsatisfied *Futures* that are evaluated in each block. Adjusting this value will cause the *Serial* mode to simulate, to some extent, differing communications network performance.
- Abort After Input File Closed** If this value is non-nil then the system will wait for the number supplied of user time units after the signal data file has been read in and closed and then it will abort the simulation. Clearly this value should be set high enough so that all processing will have been completed before the processing is terminated.
- Default AutoSave Interval** This is the number of characters to be printed to autosaving files, such as *Dribble* files before they are flushed. The use of autosaving files ensures that little data is lost in the event of a system crash.
- Filter Statistics** This flag will, when set, cause statistics printed to the statistics file to contain only those items which are likely to be of interest to the average user, as opposed to a Polygon system developer.
- Redisplay Class Pane On Creation** This flag will, when set, cause the *Class Monitor Pane* to be redisplayed whenever a new Node is created.
- Switch Dribble On When Your Run** When this option is set to *Yes* the system will execute a *Dribble On* command whenever the Run command is selected.
- Trace Messages** When this option is set to *Yes* or *Verbose* a trace message is printed in the *Message Pane*, when the system is running in its parallel mode for each message posted by the system.
- Trace Clock Ticks** When this option is set a trace message is printed out when the clock ticks.
- Trace Signal Records** When this option is set, trace messages are printed out both when a signal record is read in and when its processing has finished.
- Break Clock Ticks** When this option is set a the system executes a breakpoint when the clock ticks.
- Break Signal Records** When this option is set a breakpoint is executed when a signal record is read in.
- Trace Rules** When this option is set to *Verbose* the trace messages, which are produced when rules are being traced are extended so that the node for which the rule is firing is displayed. The default for this option is *Brief*.

Trace Node Creation When this option is set, a trace message is printed when a Node is created.

Trace Message Punting When this option is set, trace messages are printed whenever the system has to *Punt*. For more information on this issue please see Appendix Q.3.

Break Message Punting When this option is set, a breakpoint is executed whenever the system has to *Punt*. For more information on *Punting* see please see Appendix Q.3.

Scheduling Strategy This option sets a parameter which is only meaningful when executing in the *Serial* mode. The *Serial* version of Poligon attempts to replicate the semantics of the parallelism of the *Parallel* case by the relaxation of determinism in the execution of rules, for both the *Condition Parts* and the *Action Parts*. In order to support this, Poligon comes equipped with a primitive scheduler. This scheduler is parameterized in order to allow the testing of the system in the *Serial* mode with different scheduling characteristics, allowing testing for any unsuspected serial dependencies. There are two sorts of tasks that are scheduled; the *Condition* and the *Action Parts* of the rules. A data structure holds these tasks and the scheduler simply selects a task from this collection. There are four different scheduling strategies available, which can be selected by a menu. These are:

LIFO Tasks are processed in a last-in-first-out manner.

FIFO Tasks are processed in a first-in-first-out manner.

Random Tasks are picked at random from the pending events.

Immediate Tasks are processed as soon as they are generated. This is a sort of depth first behavior and is quite different from *LIFO*.

Stop At Or After Time When this option is an integer a breakpoint will be executed when the system clock reaches or passes this time.

Stop At Or After Signal Record When this option is an integer a breakpoint will be executed when the system reads or has read this signal record.

Which Instrument To Use This option lists all of the CARE instruments that are loaded and allows the user to select the desired one. The instrument called *Poligon's* is special in that it is configured into the Poligon Control frame. If an instrument other than this is selected then no instrument will be configured into the Poligon control frame and the instrument should be selected with *<System>-S*. This option is only meaningful in the *Parallel* mode.

Hardcopy Minutes	When this option is set to a number then this is interpreted as being the number of minutes to wait between performing screen dumps of the instrument. This option is only meaningful in the <i>Parallel</i> mode.
Record Message Times In Log File	When this option is selected Poligon will time the execution of the messages that are sent between processors in the <i>Parallel</i> mode and will output this as a trace to a file. This allows the user to find expensive operations.
Record Message Times In List	When this option is selected Poligon will time the execution of the messages that are sent between processors in the <i>Parallel</i> mode and will keep these in a list. This is useful if one wants to filter the traced messages programmatically.
Message Time Log File	This is the name of the message time log file which is generated by the <i>Record Message Times In Log</i> option above.
Log Messages Lasting Longer than N ms	When this option is set to a number then the messages logged by the options above are filtered so that only those messages that take longer than this value (in milliseconds) will be logged. Lots of system messages are very fast and it is usually a good idea to filter these out..
Redisplay	This causes the screen to redisplay itself. The reason for this command is that the <i>Class Monitor Pane</i> (see Section 4.3.6) does not redisplay itself when new <i>Nodes</i> are added, in the <i>Serial</i> mode. This is the case because of the frequency with which new <i>Nodes</i> are created and the the time taken by scrolling windows. Selecting this option will make the <i>Class Monitor Pane</i> display the latest results.
Run	This option causes the system to attempt to run the user's model.
Serial	This command sets Poligon to make it run in <i>Serial</i> mode. This command is equivalent to a call to the procedure (<i>Run-Polygon</i>).
Statistics	This option allows the user to print out some statistics associated with the messages sent by the system. These statistics are printed out to a file named by the <i>Statistics File</i> option of the <i>Parameters</i> command. These statistics are in four forms; the message frequencies sorted by message name, the message frequencies sorted by message frequency and the full message information sorted by message frequency and with the origin and target of the message displayed. Finally the distribution of the nodes/contexts on the sites in the system is shown.

4.3.4. The Status Pane

This is a small pane, which can be found just above the *Command Menu*. It displays information which might be of interest to the user during the execution of the model. The in-

formation differs depending on whether the system is running in its *Serial* or its *Parallel* mode. In the *Serial* mode the following information is displayed.

- The scale factor for the system clock. This is the number of simulator clock ticks which are used to represent one unit of user defined time.
- The system time, as measured in user time units.
- An estimate of the number of messages that would be posted in the *Parallel* mode.
- The number of events that the scheduler has yet to schedule. These events are the *Condition* and *Action Parts* or rules. This value is always 0 when the *Immediate Scheduler* option is being used. Following this value, separated by a vertical bar is the number of *Futures* which are still to be processed and may yet be unsatisfied.
- The scheduling strategy being used.

In the *Parallel* mode the following information is displayed.

- The scale factor for the system clock. This is the number of system clock ticks which are used to represent one unit of user defined time.
- The system time, as measured in user time units.
- The number of messages that have been posted.
- The number of agents (processes) that have been created and the number of active stack groups, separated by a vertical bar.
- The number of user signal record that have been read.
- If metering is enabled then the percentage of the metering partition that has been used.

4.3.5. The Message Pane

This pane is used in the *Parallel* mode in order to show the messages sent in the system as they are processed and any other bits of information that are likely to be of interest during the execution of the model. This pane is not used in the *Serial* mode. If CARE is not loaded then this pane is not configured into the Polygon frame.

4.3.6. The Class Monitor Pane

This is a pane which shows the items on the *Blackboard*. They are displayed as lines of text. The type of Node is on the left; *Superclass*, class or normal Node and on the right is the symbolic name of that Node. The pane is a standard scrolling window so you get the normal scroll bar on the left. The items in the window are mouse sensitive. The three mouse buttons induce different actions, which are documented in the who line. Their actions are as follows:

Left	This generates a menu which allows the user to select a number of trace and break options for this Node as a whole. Trace and break points can be set for both <i>Discard</i> and <i>Recycle</i> operations.
Meta-Left	This option is much like the plain <i>Left</i> option. It generates a menu which allows the user to select trace and break options for the <i>Instances</i> of this Node if it is a class Node. Trace and break points can be set for both <i>Discard</i> and <i>Recycle</i> operations.
Middle	This causes the selected object on to be inspected in the inspector.
Meta-Middle	This causes the values of all of the instance variables of the selected Node to be displayed. They are printed out in the

Right

Lisp window on the right of the frame. The values of the instance variables are printed out after their names.

This generates a menu which allows the user to select a number of trace and break options associated with the *Fields* of this Node. The menu is a choice box type of menu. All of the fields of the Node are listed on the left and the types of trace or break option head the choice box columns. Trace and break points can be set to look at any of the *Fields* in the relevant Node. The trace and break points can be set so that they can be activated when the selected *Field* is read from, written to, has an *Event* happen to it or is *Inherited*.

Meta-Right

This option is much like that for the plain right button only it allows you to set trace and break points for the fields of all of the *Instances* of the selected Node if it is a class Node.

4.3.7. The Rule Monitor Pane

This pane is much like the *Class Monitor Pane*. It is a scrolling window with mouse sensitive items. In this case the items in the window represent the *Knowledge Sources* and their rules. The rules are indented to indicate their membership of a *Knowledge Source*. Just as is the case with the *Class Monitor Pane* the buttons on the mouse will induce different responses, when clicked over one of these items. Their actions are as follows:

Left

Pressing this button will cause a menu to appear, which allows the user to select trace and break options, which are associated with a given rule or *Knowledge Source*, rather than Node or class as is the case on the *Class Monitor Pane*. *Knowledge Sources* do not have any real meaning at run-time so the *Knowledge Sources* mentioned here are taken to be representations for all of the rules, which are mentioned as being members of that *Knowledge Source* at compile time. It is assumed that the rule writer will group his rules into functional units at compile time and these units may also be of interest in debugging at run-time. It is therefore not meaningful to set trace and break points for *Knowledge Sources* themselves. Thus selecting a *Knowledge Source* will cause the selected trace and break options to be set for all of the Rules associated with that *Knowledge Source*. A number of options are available here.

Trace and break points can be set on the *When Parts* of rules. This means that a trace or break will happen if the *When Part* attempts to fire. It should be noted that this will happen before the test for the *When Part* is made.

Trace and break points can be set on the *If Parts* of rules. These are encountered when the *When Part* of the rule has evaluated to non-*nil*. The trace or break is executed before the *If Part* test is made.

Trace and break points can be set on the *Select Parts* of rules. These are encountered when the *If Part* of the rule has evaluated to non-*nil*. The trace or break is executed before the *Select Part* is evaluated.

Trace and break points can be set on the *Action parts* of rules. This is encountered after the *When Part* and the *If Part*

have both evaluated to non-*nil*. The trace or break is executed before any action in the *Action Part* has happened.

It is possible to trace the failure of the *If Parts* and *When Parts* of rules. It is often useful, when debugging *Models*, to find out why a rule did not fire. To this end the *Compiler* transforms the expressions which make up the conditions for a rule so that all of the operands for top level *And* expressions are separated out. Thus, at least conceptually, "A and B and (C or D)" is transformed into [A B [or C D]]. From this point the *Compiler* generates code which causes these conditions to be executed in sequence, with suitable short circuit evaluation in the event of a failure of one of the operands. From the trace and break menu it is possible to cause a trace or break point to be executed in the event of a condition failing after a given clause. Thus if one suspected that the clause [or C D] was not working correctly a trace or break could be put on this rule which would be executed if the expression failed on or after clause three, the one in question. Setting this option to 0 will always give a trace/breakpoint when the condition fails.

One of the things that can go wrong with rules is the expressions for the definition parts might have bugs. In the event of breaking in a rule it is not too easy to get the values of the *Definitions*. A facility has been provided so that the *Definitions* for a given rule can be forced to be evaluated and printed out. These can be forced and printed out when the *When Part* fires, when the *If Part* or when the *Action Part*, the *Otherwise Part* or the *Timeout Part* fires.

It should be noted that the default configuration for this menu causes a trace point to be set on the *Action Part* of all rules.

Middle

This button is much like the middle button for the *Class Monitor Pane*. This causes the rule or *Knowledge Source* clicked on to be inspected in the inspector.

Meta-Middle

The internal structure of the item on which you clicked is displayed in the Lisp pane on the right.

Right

Pressing this button will cause a menu to appear, which allows the user to set whether the rule or *Knowledge Source* in question is active or not. If a *Knowledge Source* is selected then it is equivalent to selecting every rule in the *Knowledge Source* with the same value.

4.3.8. The Graphics Pane

This pane is used to display *Graphics* produce by the user's model. If the *Graphics* subsystem is not loaded this pane will not be configured into the Polygon control frame. For information on the use of the *Graphics* subsystem the user is encouraged to read its associated documentation.

A. A simple way to make a Polygon model

This section describes briefly a simple set of actions, which will allow a new user to make a Polygon model.

Type in the *Class Declarations* into a file called *Classes.Polygon*.

This file should have a *Mode Line* (the top line in the file) which specifies the following:

```
;;;-*-Mode:Polygon; Package:Polygon-User; Base:10.-*-
```

This *Mode Line* could, if required, contain a font specification.

Type any functions in a file called *Functions.Polygon*. This should also have the *Mode Line* specified above, if the functions are written in Polygon, or it should be of type *Lisp* with the appropriate *Mode Line* if it is in Lisp.

Type in the rules into a file called *Rules.Polygon*. This file should have a *Mode Line* like the one specified above.

Compile and load the *Class Declarations* through the Polygon *Compiler*, if appropriate, and the Common Lisp compiler. This can be done most easily by the use of the Zmacs command *Compile and Load File*.

Compile and load the functions. This should be done as for the classes mentioned above. Compile and load the rule base. This should be done as for the functions and *Class Declarations* mentioned above.

You should now be ready to execute your model so you should type:

```
(Run-Polygon)      ;;; Just run in the serial mode for now.
```

and then hit the Run button.

B. Some Example Functions written in L100 or Poligon

The standard example of "Factorial"

```
Define Factorial (a-number : Integer)
  Documentation
    "This function computes the factorial of"&
    "a number"
  If a-number = 1 Then 1
  Else a-number * Factorial(a-number - 1)
  EndIf
EndDefine
```

This example shows the use of sundry infix *Operators*, a type denotation for the argument and extended *Strings* for the documentation *String*.

A function to construct a new *List* from head and tail components that it calculates.

```
Define Make-a-new-list
  Documentation "This function makes a new list"
  let the-head = a-function-to-determine-the-head()
  &let the-tail = a-function-to-determine-the-tail()
  in
    the-head > the-tail
  EndLet
EndDefine
```

This example shows the use of the *Let* form, the application of parameterless functions and the definition of a parameterless function. An empty *Parameter List* could have been provided before the documentation *String*.

A function that takes a *List* of *Lists* and returns a sorted *List* of *Lists*, sorted on the sum of the first and second elements of each *List*.

```
Define Sort-a-list-of-lists (a-list-of-lists)
  Sort(a-list-of-lists,
    λ(sublist-1, sublist-2)
      Documentation "Compare two lists"
      sublist-1•First + sublist-1•Second
      < sublist-2•First + sublist-2•Second
    Endλ)
EndDefine
```

This function shows the definition of a function without a documentation *String* and a *Lambda* expression taking two arguments with a documentation *String*. The postfix function application *Operator* "•" is used to extract the elements from the sublists.

A procedure that assigns 42 to a global variable:

```
Variable Count ← nil

Define An-updater ()
```

```

Documentation
"This procedure updates the value of the global "&
"variable 'Count'."
42 → Count
EndDefine

```

This procedure shows the declaration of a global variable, which is initialized to nil, an extended documentation *String* and the use of the assignment *Operator*.

A function to construct a *List* with two variable components, passed as arguments and a number of constant components. The first argument is inserted into the *List* as an element. The second is expected to be a *List*, which is appended into the *List*.

```

Define Make-up-a-mixed-list (Component-1, Component-2)
  [1 2 3 ↑Component-1 4 5 6 ↓Component-2 7 8 9 10]
EndDefine

```


C. An Example Poligon Model

This appendix contains an example Poligon model. First the model will be explained in small chunks and then a complete listing will be given.

The model itself is, of course, trivial but shows some of the features of the language. It concerns a farm, where there are workers, presumably with binoculars and radios, reporting on the locations of animals. The model contains an extensive set of *Class Declarations* in order, to show how the *Blackboard* might be structured for such a problem, a full example of the *Data Input* component definition to handle reports of cattle and sheep and a *Knowledge Source* with one rule, that deals with the input from the observers on the farm.

C.1 An Example Model explained in pieces

The model being described here must reside in two files. The first one contains definitions of data *Structures*, in this case only one. The second contains the remainder of the model. This is because of the "declare before use" requirements of Poligon, which requires that data *Structure* definitions must be both compiled and loaded before the functions that they define can be used.

C.1.1 The Data Structures File

This section contains the specification of the data *Structure* mentioned in this model. This data *Structure* is not used to any significant degree in the example model but is shown here both to show how such a definition might be made and to stress the requirement that these declarations be in a separate file. The code is as follows.

```
;;; -*- Mode:Poligon; Package:Poligon-User; Base:10 -*-  
  
Structure Coordinate_pair Fields x_coord, y_coord
```

This example shows the *Mode Line* used in the file and the definition of a data *Structure* type called *Coordinate_pair*, which has two component fields called *x_coord* and *y_coord*. The result of this is that access functions called *x_coord* and *y_coord* will be defined, which can extract the values of the relevant fields, a *Structure* creating function called *Make_Coordinate_pair* (&Key *x_coord* *y_coord*) and a *Structure* type predicate called *Is_a_Coordinate_pair*.

C.1.2 The Class Declarations for the Example Model

This section shows the *Class Declarations* used by the example model. The world seen by the model is seen as being made up a number of broad categories of objects, which are specialized. For instance, all animals on the farm are thought of as belonging to the class *Farmyard-Animals*. From this class they inherit the attributes *Weight*, *Colour*, *Serial-Number* and *Position*. No animals of the class *Farmyard-Animals* are ever created. This class is used as an *Abstract Superclass* in order to define more specialized forms of animal.

```
;;; -*- Mode:Poligon; Package:Poligon-User; Base:10 -*-  
Class Definitions For Model "My Farm-Yard Model" :  
  Class Farmyard-Animals :  
    Fields : Weight  
            Colour
```

Serial-Number
Position

The another major form of object as seen by the model is a *Sighting*. This is a report from a farm worker concerning the location of one of the animals. The class *Sighting* is also an *Abstract Superclass*, since all sightings will be of more specialized *Classes* of sighting, which will be assumed to contain information about the sort of animal seen. However all sightings have the common attributes of *Colour*, *Serial-Number* which is painted on the side of the animal, and *Position*, which will be a map reference of the type *Coordinate pair*.

Class Sighting :
Fields : Colour
Serial-Number
Position

The animals seen are not just individual animals wandering about, they are always members of some larger collection; flocks in the case of sheep and herds in the case of cattle. The general characteristics of these entities are represented on the *Blackboard* by the *Abstract Superclass Collection*. This *Superclass* confers the property *General-Location*, the mean position of the animals in that collection on the *Instances* of its *Subclasses*.

Class Collection-Of-Things :
Fields : General-Location

Now comes a selection of other class declarations, which specify other characteristics of animals in which the model writer might be interested. *Mammals* are defined to have four legs by default, *Birds* are defined to have only two and all edible animals are to have attributes that allow the model in some way to reason about the price per pound of their meat and about wines that might be drunk, whilst the meat is being consumed.

Class Mammals :
Fields : Number-of-legs : 4
Class Birds :
Fields : Number-of-legs : 2
Class Edible-Animals :
Fields : Price-per-pound
Suitable-wines

Now the model will define the animals in which it is really interested; *Sheep* and *Cattle*. Both of these *Classes* of *Nodes* are *Farmyard-Animals*, *Edible-Animals* and *Mammals* as well as being themselves. *Sheep* have the attribute *Thickness-of-wool*, which is specific to *Sheep* (this farm has neither goats, angora rabbits, guanaco, alpaca nor vicuña) and *Cattle* have the attribute *Milk-Output* (this is a non-breeding dairy farm).

Class Sheep :
Superclasses : Farmyard-Animals, Edible-Animals,
Mammals
Fields : Thickness-of-wool
Class Cattle :
Superclasses : Farmyard-Animals, Edible-Animals,
Mammals
Fields : Milk-Output

This completes the definition of the *Classes* that define the animals on the farm. Next the *Classes* of *Collection-Of-Things* will be defined. *Flocks* and *Herds* are both simply *Collections-Of-Things*. They have the same attributes but are distinct.

```
Class Flocks :  
    Superclasses : Collection-Of-Things  
Class Herds :  
    Superclasses : Collection-Of-Things
```

Now the *Classes* of *Sighting* will be defined. Like *Flocks* and *Herds* the different types of *Sighting* have the same attributes but are represented as distinct types.

```
Class Sheep-Sighting :  
    Superclasses : Sighting  
Class Cattle-Sighting :  
    Superclasses : Sighting
```

Finally the *Root* class and *Input Handler Class*, called *Input-Handler* are defined. These are simply the minimal forms of the *Root* and *Input Handler* definitions.

```
Class Root :  
    Fields :  
Class Input-Handler :  
    Metaclasses : Input-Handler-Class-Mixin  
    Superclasses : Input-Handler-Mixin
```

C.1.3 Initialisation in the Example Model

Initialisation is very simple in this model. There is none of it. The system requires, however, that the user states that he has no need of any special *Initialisation* code. This is done by the following.

```
.i.User Defined Initialisation;Initialisation :
```

If the model had a need to open a debug logging file or such like this is where it would be done.

C.1.4 The Data Input declaration for the Example Model

The *Data Input* definition consists of the definition of a function and a procedure. These decode the input data and put it onto the *Blackboard*. The input data is assumed to be a list of the following form.

```
(Timestamp Record-Type Serial-Number Colour Position-x  
Position-y)
```

Where *Timestamp* is an integer denoting the time in the user's units, *Record-Type* is a :Keyword denoting the type of sighting (sheep or cow), *Serial-Number* is some representation of the serial number of the animal, *Colour* is some representation of the colour of the animal and *Position-x* and *Position-y* are numbers denoting the X and Y coordinates of the sighted animal.

A function must be defined which extracts the timestamp from the signal data. This function must be called *Time-of-Input-Record* and, for this model is defined below.

```

Define Time-of-Input-Record (Record)
  Record•The-First
EndDefine

```

Next the model must define the way in which the signal data is to be put onto the *Blackboard*. This is done in the procedure, which the model must define, called *Input-Procedure*. This procedure is defined below.

```

Define Input-Procedure
  (Record, Timestamp, The-Input-Handler)
  Ignore(Timestamp, The-Input-Handler)
  Case Record•The-Second Of
  Choice :Sheep :
    New Instance of Sheep-Sighting
    Initialisation :
      Serial-Number ← Record•The-Third
      Colour        ← Record•The-Fourth
      Position      ← Make_Coordinate_pair
                      (:x_coord , Record•The-Fifth,
                      :y_coord , Record•The-Sixth)
  Choice :Cow :
    New Instance of Cattle-Sighting
    Initialisation :
      Serial-Number ← Record•The-Third
      Colour        ← Record•The-Fourth
      Position      ← Make_Coordinate_pair
                      (:x_coord , Record•The-Fifth,
                      :y_coord , Record•The-Sixth)
  Otherwise : Ferror(nil, "Illegal input record type.")
  EndCase
EndDefine

```

As can easily be seen, this procedure cases on the record type and creates an *Instance* of the class *Sheep-Sighting*, when a sheep has been seen and an *Instance* of the class *Cattle-Sighting* when a cow has been seen.

C.1.5 Application Dependent Functions in the Example Model

In this model there will be the need to determine whether one Node is near to another. For instance, it is necessary to determine when a sheep is near to a flock, so a criterion for this nearness is defined here by the predicate *Is-Near*. This is true if the coordinates supplied to it are "near to one another".

```

Constant Distance_Tolerance = 100

Define within-tolerance (distance_1, distance_2)
  Documentation
    "Is non-nil if the two distances are within a defined
    tolerance"
    Abs(distance_1 - distance_2) < Distance_Tolerance
  EndDefine

Define Is-Near (coordinates_1, coordinates_2)
  Documentation

```

```

    "Takes two coordinates and is non-nil if they are near
    one another"
    within-tolerance (coordinates_1•x_coord,
                      coordinates_2•x_coord)
    And within-tolerance (coordinates_1•y_coord,
                          coordinates_2•y_coord)
EndDefine

```

C.1.6 The Knowledge Source in the Example Model

This example model has only one *Knowledge Source*. A real model would, of course, have more but this one should serve to explain how the language is used to achieve the goals of the model writer.

The *Knowledge Source* begins with the *Knowledge Source* declaration itself and *Knowledge Source* level *Definitions*. In this case the *Knowledge Source*, which is being defined is called *Correlate-Sightings-With-Existing-Animals* and makes one *Knowledge Source* level *Definition*; *Sheep-with-the-same-number*.

```

Knowledge Source :
    Correlate-Sightings-With-Existing-Animals
Definitions : Sheep-with-the-same-number =
    Subset Of Sheep For Which
    Element•Serial-Number
    = The-Serial-Number

```

In this case the *Definition* of the name *Sheep-with-the-same-number* would be seen in all rules in this *Knowledge Source*. The value of *Sheep-with-the-same-number* is a *Bag* of all of the *Subsystems* of the Node *Sheep* (in fact the class of *Sheep*) that has a *Latest* value of its *Serial-Number Field* equal to the serial number denoted by *The-Serial-Number*. The latter is the special identifier denoting the name of the *Field* upon which the rule shown in Section C.1.7 hangs.

C.1.7 The Rule in the Example Model

The example model has only one rule. This rule, however, does quite a lot. It fires when new *Sheep-Sightings* are created and their *Serial-Number Fields* are set.

```

Rule : Correlate-Sheep-Sightings-With-Existing-Sheep
Class : Sheep-Sighting
Field : Serial-Number

```

The rule is called *Correlate-Sheep-Sightings-With-Existing-Sheep*, is associated with every *Instance* of the class *Sheep-Sighting* and hangs on the *Field* called "*Serial-Number*". Within the rule the names *The-Sheep-Sighting* and *The-Serial-Number* will represent the Node which has had an event on its *Serial-Number Field* and the new value in the *Field Serial-Number* on that Node respectively. The name *The-Serial-Number* is used by the *Definition* made in the *Knowledge Source* header in Section C.1.6.

This rule itself comes in four main parts; the *Rule Header* shown above, the *Condition Part*, the *Action Part* and the *Otherwise Part*.

Having found a new *Sheep-Sighting Instance* the rule fires its *Action Part* if there is already a sheep *Instance* with the same serial number. This is taken to be a new sighting of the

same sheep. If there is no sheep with the same serial number then the *Otherwise Part* fires, creating a new sheep *Instance* for the sighting.

C.1.8 The Condition Part of the Rule in the Example Model

The *Condition Part* of the example rule has no test in its *When Part*. This means that there is nothing more that can be deduced about whether the rule should be allowed to fire or not without looking at *Nodes* other than the *Sheep-Sighting*, which caused the rule to fire. The *If Part* will be true if there are any sheep with the same serial number as the serial number of the sheep sighting. The sheep with the same serial number are held in *Sheep-with-the-same-number*, the bag defined in the *Definitions* in Section C.1.6. It is assumed by this model that there will only ever be one sheep with the same serial number. Clearly a more sophisticated model would try to cope with the ambiguity caused by having a number of sheep with which the sighting correlated.

```
Condition Part :  
  When : t  
  If   : Sheep-with-the-same-number*  
        Number-of-Elements ≠ 0
```

C.1.9 The Action Part of the Rule in the Example Model

If the *Condition Part* of the example model is true then it is known that there is already an *Instance* of the class sheep, with which the sighting has been correlated. The rule must now update the position of the sheep on the basis of the new sighting. Because it is known that this sighting *Node* has now been finished with, since all information will have been extracted from it that is needed it can be *Recycled*. However in order to be able to *Recycle* it in *Parallel* with the *Update* to the sheep *Node*, which is to have the new position, the value from the sighting's position is read before either of these side effects can happen. This is done by *Forcing a Definition*, which reads the value of this *Field*. Once this has been done the *Update* to the sheep *Node*, which is one of the elements in the *Bag* denoted by *Sheep-with-the-same-number* defined at the *Knowledge Source* level in Section C.1.6 can happen. In this case the *Position Field* of the sheep *Node* is *Updated* with the *Modify* operator so that a record is kept of all of the places that this sheep has been. In *Parallel* with this the *Sheep-Sighting Node* is *Recycled*.

```
Action Part :  
  Definitions :  
    Position-of-the-sighting =  
      The-Sheep-Sighting•Position  
  Force : Position-of-the-sighting  
  Changes :  
    Change Type      : Update  
    Updated Node     :  
      Sheep-with-the-same-number•An-Element  
    Updated Fields :  
      Position ← Position-of-the-sighting  
  Changes :  
    Change Type      : Recycle  
    Updated Node     : The-Sheep-Sighting
```

C.1.10 The Otherwise Part of the Rule in the Example Model

The *Otherwise Part* of the example rule is invoked if there is a new *Sheep-Sighting*, which does not match with any existing sheep. To account for this sighting a new *Instance* of the class *Sheep* is to be created. Sheep are always associated with flocks so the new sheep must have as its "parent" flock - the flock that it is near to, or failing that a new flock.

The *Otherwise Part* of the example rule makes a number of *Definitions*. These are as follows.

- Position-of-sighting** The value of the *Position Field* of the sighting. This is forced in order to allow the *Recycling* of the sighting in the same way that it is in Section C.1.9.
- Colour-of-the-sighting** The value of the *Colour Field* of the sighting. This is forced in order to allow the *Recycling* of the sighting in the same way that the *Position Field* is in Section C.1.9.
- Serial-number-of-the-sighting** The value of the *Serial-number Field* of the sighting. This is forced in order to allow the *Recycling* of the sighting in the same way that the *Position Field* is in Section C.1.9.
- Flocks-which-match-new-sheep** A *Bag* containing all of the flocks which are near the sighting (satisfy the *Is-Near* predicate defined in Section C.1.5).
- Flock-For-New-Sheep** If there is a flock in the *Flocks-which-match-new-sheep Bag* then that flock (it is assumed that only one flock matched this test). If there is no matching flock then a new *Instance* of the flock class, whose general position is the same as the position of the sighting.
- New-Sheep** The new *Instance* the class *Sheep*, whose *Supersystems* are the Node, which represents the class *Sheep* and the flock of which this sheep is to be a member. The *Fields* of this Node are initialized to have the values of the *Fields* in the sighting.

The code for the *Otherwise Part* is as follows.

```
Otherwise Part :
  Definitions :
    Position-of-sighting =
      The-Sheep-Sighting•Position
    Colour-of-the-sighting =
      The-Sheep-Sighting•Colour
    Serial-number-of-the-sighting =
      The-Sheep-Sighting•Serial-Number
    Flocks-which-match-new-sheep =
      Subset of Flocks Which Satisfies
         $\lambda(a\text{-flock})$ 
          is-near(The-Sheep-Sighting•Position,
                  a-flock•General-Location)
      End $\lambda$ 
    Flock-For-New-Sheep =
      If Flocks-which-match-new-sheep•
        Number-of-Elements = 0
      Then New Instance of Flocks
```

```

        Initialisation :
            General-Location ←
                Position-of-sighting
        Else Flocks-which-match-new-sheep
            An-Element
        EndIf
    New-Sheep =
        New Instance of Sheep
        Subsystem Of : Flock-For-New-Sheep, Sheep
        Initialisation :
            Serial-Number ←
                Serial-number-of-the-sighting
            Colour          ← Colour-of-the-sighting
            Position        ← Position-of-sighting
    Force : Position-of-sighting, New-Sheep
    Changes :
        Change Type      : Recycle
        Updated Node     : The-Sheep-Sighting

```

C.2 A Complete Listing of the Example Model

The following is a complete listing of the example model explained in detail in Section C.1.

```

;;; -*- Mode:Poligon; Package:Poligon-User; Base:10 -*-
Structure Coordinate_pair Fields x_coord, y_coord

;;; -*- Mode:Poligon; Package:Poligon-User; Base:10 -*-
Class Definitions For Model "My Farm-Yard Model" :
    Class Farmyard-Animals :
        Fields : Weight
                Colour
                Serial-Number
                Position
    Class Sighting :
        Fields : Colour
                Serial-Number
                Position
    Class Collection-Of-Things :
        Fields : General-Location
    Class Mammals :
        Fields : Number-of-legs : 4
    Class Birds :
        Fields : Number-of-legs : 2
    Class Edible-Animals :
        Fields : Price-per-pound
                Suitable-wines
    Class Sheep :
        Superclasses : Farmyard-Animals, Edible-Animals,
                        Mammals
        Fields : Thickness-of-wool
    Class Cattle :
        Superclasses : Farmyard-Animals, Edible-Animals,
                        Mammals
        Fields : Milk-Output

```



```

Class Flocks :
    Superclasses : Collection-Of-Things
Class Herds :
    Superclasses : Collection-Of-Things
Class Sheep-Sighting :
    Superclasses : Sighting
Class Cattle-Sighting :
    Superclasses : Sighting
Class Root :
    Fields :
    Class Input-Handler :
        Metaclasses : Input-Handler-Class-Mixin
        Superclasses : Input-Handler-Mixin
Initialisation :

Define Time-of-Input-Record (Record)
    Record•The-First
EndDefine

Define Input-Procedure
    (Record, Timestamp, The-Input-Handler)
    Ignore(Timestamp, The-Input-Handler)0
    Case Record•The-Second Of
    Choice :Sheep :
        New Instance of Sheep-Sighting
        Initialisation :
            Serial-Number ← Record•The-Third
            Colour        ← Record•The-Fourth
            Position       ← Make_Coordinate_pair
                           (:x_coord , Record•The-Fifth,
                           :y_coord , Record•The-Sixth)
    Choice :Cow :
        New Instance of Cattle-Sighting
        Initialisation :
            Serial-Number ← Record•The-Third
            Colour        ← Record•The-Fourth
            Position       ← Make_Coordinate_pair
                           (:x_coord , Record•The-Fifth,
                           :y_coord , Record•The-Sixth)
    Otherwise : Ferror(nil, "Illegal input record type.")
    EndCase
EndDefine

Constant Distance_Tolerance = 100

Define within-tolerance (distance_1, distance_2)
    Documentation
    "Is non-nil if the two distances are within a defined
    tolerance"
    Abs(distance_1 - distance_2) < Distance_Tolerance
EndDefine

Define Is-Near (coordinates_1, coordinates_2)
    Documentation

```

```

    "Takes two coordinates and is non-nil if they are near
    one another"
        within-tolerance (coordinates_1.x_coord,
                           coordinates_2.x_coord)
    And within-tolerance (coordinates_1.y_coord,
                           coordinates_2.y_coord)
EndDefine

```

```

Knowledge Source :
    Correlate-Sightings-With-Existing-Animals
Definitions : Sheep-with-the-same-number =
                Subset Of Sheep For Which
                Element.Serial-Number =
                The-Serial-Number
Rule : Correlate-Sheep-Sightings-With-Existing-Sheep
Class : Sheep-Sighting
Field : Serial-Number
Condition Part :
    When : t
    If   : Sheep-with-the-same-number •
            Number-of-Elements ≠ 0

Action Part :
    Definitions :
        Position-of-the-sighting =
            The-Sheep-Sighting • Position
    Force : Position-of-the-sighting
    Changes :
        Change Type      : Update
        Updated Node      :
            Sheep-with-the-same-number • An-Element
        Updated Fields :
            Position ← Position-of-the-sighting
    Changes :
        Change Type      : Recycle
        Updated Node      : The-Sheep-Sighting
Otherwise Part :
    Definitions :
        Position-of-sighting =
            The-Sheep-Sighting • Position
        Colour-of-the-sighting =
            The-Sheep-Sighting • Colour
        Serial-number-of-the-sighting =
            The-Sheep-Sighting • Serial-Number
        Flocks-which-match-new-sheep =
            Subset of Flocks Which Satisfies
            λ(a-flock)
                is-near(The-Sheep-Sighting • Position,
                        a-flock • General-Location)
            Endλ
        Flock-For-New-Sheep =
            If Flocks-which-match-new-sheep •
                Number-of-Elements = 0

```

```

Then New Instance of Flocks
  Initialisation :
    General-Location ←
      Position-of-sighting
Else Flocks-which-match-new-sheep•An-Element
EndIf
New-Sheep =
  New Instance of Sheep
    Subsystem Of : Flock-For-New-Sheep, Sheep
      Initialisation :
        Serial-Number ←
          Serial-number-of-the-sighting
        Colour      ← Colour-of-the-sighting
        Position    ← Position-of-sighting
Force : Position-of-sighting, New-Sheep
Changes :
  Change Type      : Recycle
  Updated Node     : The-Sheep-Sighting

```

D. How to load the L100 system and how to start it up

The L100 language is loaded by the command:

```
(Make-system 'L100 :Silent :Noconfirm :Nowarn)
```

This gives access to the basic L100 language and the L100 compiler. The L100 language is loaded automatically when Poligon is loaded. This is shown, therefore, mainly for reasons of completeness.

The user need only know about three procedures and functions which provide access to the compiler. These are as follows:

Parse (file_name, language_name) Given a filename and a *Keyword* denoting the name of the language to be used by the compiler the compiler parses the contents of the file and generates suitable compiled code. This code is put into a file of the same name but with the file type "Lisp". Once the output file has been written out the file is loaded into Zmacs for you. The value of a call to this function is a record of type *Code structure*. This code structure can normally be ignored but it can be of significance if you want to do incremental compilation and yet carry forward some compiler context.

Parse_in_Context(filename, a_code_structure, language) This procedure is just like *Parse* only it allows the user to pass a code structure to it as an argument. It is by this means that the user can achieve modular compilation.

Parse_a_list (a_list, old_code, language) This function takes a *List* of tokens, a code structure and a *Keyword* denoting the name of the language and parses the *List* in the context of the code structure. The *List* can be a *Lazy List*.

More documentation for these can be found in the L100 source code, which is extensively documented.

Once the compilation process has happened you then have to compile the Lisp code for yourself unless the compilation is being done inside Zmacs or by the use of *Make-system/DefSystem*.

E. Data Structures and Types

A number of new *Types* have been defined in Poligon. These and the functions used to manipulate them are described in this section.

The new data structures supported by Poligon are *Bags*, *Sets* and *Lazy Lists*. Most of the same operations apply to *Bags*, *Sets*, normal lists and *Lazy Lists* and these are listed in Section E.6, a joint section below. Those functions, which are exclusive to a given type will be listed and explained here. All of the above data types are types of *Collection*.

E.1 Types defined in Poligon

The following types are defined in Poligon and may either be used by the programmer or will be seen by the programmer during debugging.

Collection	The type of any collection. This includes Nil, lists, <i>Bags</i> , <i>Sets</i> and <i>Lazy Lists</i> and any system defined internal <i>Collection</i> data types. The operations on <i>Collections</i> are shown below.
Functional-Value	The type to which all functions and functional objects belong. This includes Poligon's internal representation of functional objects.
Future	The type that denotes promises for values.
Link-Cell	The type of one of the ends of a <i>Link</i> .
Multi-Future	The type that denotes a promise for a <i>Collection</i> of values.
Multiple-Values	The type used to denote multiple values returned by a function or <i>Definition</i> .
Non-Nil-Collection	This type is just like the type <i>Collection</i> only it excludes Nil.
Ordered-Collection	This type is any type of <i>Collection</i> which has a meaningful order. <i>Lists</i> are <i>Ordered Collections</i> .
Quoted-Form	This is the type of anything which is represented internally by the form (Quote <form>).
Type-Specifier	This is the type of any legal type specifier. Examples of type specifiers might be <i>Cons</i> , <i>Bag</i> and <i>Integer</i> .

E.2 Functions for the manipulation of types

The following functions and operators are defined in Poligon in order to manipulate types. These are defined because of the fact that the type of a value that the user is manipulating can vary depending on whether it is being seen by a strict operator or not.

value Is-Of-Type type ->	<i>T</i> if <i>value</i> appears to be of type <i>type</i> when seen by a non-strict operator.
value Is-Strictly-Of-Type type ->	<i>T</i> if <i>value</i> appears to be of type <i>type</i> when seen by a strict operator.
Non-Strict-Type-Of(value) ->	The type of <i>value</i> when it is seen by non-strict operators.
Strict-Type-Of(value) ->	The type of <i>value</i> when it is seen by strict operators.

E.3 Lazy Lists

Is-a-lazy-list(a-list) -> T if a-list is either a list or a *Lazy List*, otherwise *nil*.
The *Strict-Type-Of* of a *Lazy List* will be *Cons*.

Is-a-lazy-list-tail(a-tail) -> T if the value is the *Tail* of a *Lazy List*, i.e. is that data structure, which is used by *Tail* to compute more elements, otherwise *nil*.

Make-into-lazy-list(a-function, a-starting-element) -> A *Lazy List*, whose *Head* is *a-starting-value* and whose successive tails will be evaluated by calling a function with the preceding value. Thus *a-starting-value* will be used to calculate the second element of the list and the second will be used to evaluate the third and so on. *a-function* will be called with *:Get-Next-Value* as its first argument and the last value as its second argument. This is the case so that, if necessary, *a-function* can be an object, which will be sent a *:Get-Next-Value* message with the preceding value as its argument.

Rest(a-list) [Cdr] -> The *Tail* of a list. This will not cause the evaluation of the *Tail* of a *Lazy List*. This should be used with caution. The *Rest* of a *Lazy List Tail* will give an error and calls to *Tail* for a *Lazy List Tail* can only be made to the *Cons* cell, which has the *Lazy List Tail* as its *CDR*, since the *CAR* is used as the starting value.

E.4 Bags

Bag(&Rest, elements) -> Creates a *Bag* containing *elements*.

Is-a-Bag(a-bag) -> T if a *Bag* is a *Bag*, otherwise *nil*. The *Strict-Type-Of* of a *Bag* is *Bag*.

E.5 Sets

Set(&Rest, elements) -> Creates a *Set* containing *elements*.

Is-a-Set(a-set) -> T if a *Set* is a *Set*, otherwise *nil*. The *Strict-Type-Of* of a *Set* is *Set*.

E.6 Collections: Bags, Sets, Lists and Lazy Lists

All-Elements(a-collection) -> a list containing all of the elements in the *Collection*. This value contains only determined elements and the function will wait until all of the elements have been determined.

An-Element(a-collection) -> Values denoting an element in the *Collection* and the second denoting the *Collection* left over, i.e. a *Collection* with that element not present. This function always returns *Determined Elements* and will wait until a *Determined Element* is free. If the *Collection* is empty then the values *:Empty* and either *:List* or a *:Keyword* denoting the type of the collection, for instance *:Bag* will be returned.

Associate(item, a-collection, &Key (Key #'Head), (Result #'Identity)) -> *Result*(The first element found in *a-collection* for which *item* = *Key*(the element in the collection)). If there are no items that match then *nil*.

collection As-Type type -> A collection, whose elements are those in *Collection* but whose type is *Type*. If *Type* is a *Collection* then a collection is returned whose type is that of *Type*.

collection Cardinality-Exceeds cardinality -> *T* if the cardinality of the collection exceeds *Cardinality*, otherwise *Nil*.

a-collection Collection-Difference another-collection -> A *Collection*, which has as its elements all of those elements that are in both *a-collection* that are not in *another-collection*. The type of the value of this function is the same as that of *a-collection*.

a-collection Collection-Intersection another-collection -> A *Collection*, which has as its elements all of those elements that are in both *a-collection* and *another-collection*. The type of the value of this function is the same as that of *a-collection*.

Collection-Is-Empty(collection) -> *T* if the collection has no elements, otherwise *Nil*.

Collection-Is-Not-Empty(collection) -> *Nil* if the collection has no elements, otherwise *T*.

a-collection Collection-Union another-collection -> A *Collection*, which has as its elements all of the elements that are in *a-collection* and all of the elements in *another-collection*. The type of the value of this function is the same as that of *a-collection*.

Copy-Collection(collection) -> A new collection with exactly the same shape as *Collection* but which is distinct.

Current-Number-Of-Elements(a-collection) -> The number of *Determined Elements* in a *Collection*.

Element-of(n, a-collection) -> an element from the *Collection* by applying *Head* to the result of applying *Tail* to it *n-1* times. This is the generic equivalent of *Nth*.

Fold(a-function, a-collection, initial-value, &Rest, any-other-parameters) -> Takes *a-function* and folds it left associatively over *a-collection*, using *initial-value* as the first argument to the first call and the *Head* as the second. Any other arguments supplied follow these.

Fold-Right(a-function, a-collection, last-value, &Rest, any-other-parameters) -> Takes *a-function* and folds it right associatively over *a-collection*, using *last-value* as the second argument to the first call and the last element in the collection as the first. Any other arguments supplied follow these.

Get-Property(a-collection, key) -> This is the *Collection* equivalent of the Common Lisp function *Get*. It works for all symbols and *Ordered-Collections*, i.e. disembodied property collections. It should be noted that this function does not have the restriction that *Get* has which required it to take a locative for disembodied property lists. Thus *Get-Property(List(:a, 1, :b, 2), :a)* -> *1*.

Head(a-collection) -> an element from the *Collection*. If there are no *Determined Elements* in the *Collection* then it will wait until there is a *Determined Element*.

Is-A-Collection(a-collection) -> *T* if *a-collection* is a *Collection*, otherwise *Nil*.

Is-An-Ordered-Collection(a-collection) -> *T* if *a-collection* is an ordered *Collection* such as a list, otherwise *Nil*.

Is-An-UnOrdered-Collection(a-collection) -> *T* if *a-collection* is an unordered *Collection* such as a *Bag*, otherwise *Nil*.

a-member Is-In a-collection -> If *a-member* is a member of the *Collection* then it returns either the *Tail* of the *Collection* which has had elements extracted from *a-collection* until a match is found. This function will wait until elements have been determined if necessary. If the element is not found then *nil* is returned.

a-member Is-Not-In a-collection -> If *a-member* is a member of the *Collection* then it returns *Nil*, otherwise it returns *T*.

Join(&Rest Collections) -> This is the *Collection* equivalent of the *Append* function. It takes a number of *Collections* and returns a new composite collection containing all of the elements of the original collections.

Make-a-Collection-of-Type(collection-or-type, &Optional (length 0), (initial-elements nil)) -> Returns a new *Collection* of type *collection-or-type* if *collection-or-type* is the name of a type of *Collection*, or of the type of *collection-or-type* if *collection-or-type* is a *Collection*. The new *Collection* has *Length* initial elements, all of which are *initial-elements*.

Map-Over-A-Collection(a-function, a-collection, &Rest, any-other-parameters) -> A *Collection* denoting the values, which result from applying *a-function* to all of the elements in the *Collection*, using the element from the *Collection* as the first argument to *a-function* and *any-other-parameters* following.

Number-of-Elements(a-collection) -> values denoting the number of elements in the *Collection* and a *:Keyword* denoting the type of the collection. If the *Collection* is a *Bag* then the value is *:Bag* is returned. If it is a list ending in *nil* then the value is *:List*. If it is a chain of *Cons*es not ending in *nil* then the value is *:Lazy-List*, if the *Tail* is a *Lazy list Tail* or *:Cons* if it is not a *Lazy List*.

Sort-Collection(a-collection, predicate, &Key Key) -> This is just like the system function *Sort* only it works for any *Ordered-Collection* and returns a sorted *copy* of the original.

Tail(a-collection) -> a *Collection* which does not contain the element that would have been the value of *Head(a-collection)*.

The-Eighth(a-collection) -> A synonym for *Element-Of(7, a-collection)*.

The-Excluded-Subset(a-predicate, a-collection, &Rest other-args) -> A *Collection* like the argument but containing only those elements which fail the application *Apply-function(a-predicate, element, other-args)*.

The-Fifth(a-collection) -> A synonym for *Element-Of(4, a-collection)*.

The-First(a-collection) -> A synonym for *Element-Of(0, a-collection)* and for *Head(a-collection)*.

The-Fourth(a-collection) -> A synonym for *Element-Of(3, a-collection)*.

The-Last(a-collection) -> Returns the last element of *a-collection* for any *Ordered-Collection*.

The-Ninth(a-collection) -> A synonym for *Element-Of(8, a-collection)*.

The-Second(a-collection) -> A synonym for *Element-Of(1, a-collection)*.

The-Seventh(a-collection) -> A synonym for *Element-Of(6, a-collection)*.

The-Sixth(a-collection) -> A synonym for *Element-Of(5, a-collection)*.

The-Subset(a-predicate, a-collection, &Rest other-args) -> A
Collection like the argument but containing only those elements which satisfy the application *Apply-function(a-predicate, element, other-args)*.

The-Tenth(a-collection) -> A synonym for *Element-Of(9, a-collection)*.

The-Third(a-collection) -> A synonym for *Element-Of(2, a-collection)*.

Uniqueise(a-collection) -> Returns a collection of the same type as *a-collection* but which has no duplicate elements.

a-collection With an-element -> a *Collection* which contains an extra instance of the value *an-element*.

a-collection Without a-member -> a *Collection* which has any instances of *a-member* removed.

F. The L100 Zmacs interface

L100 and languages based on L100 can be interfaced to Zmacs in a number of ways. These are described below.

There is an L100 Zmacs major mode. All L100 code should have L100 has the mode line attribute, or should use the Zmacs major mode defined for the L100 derivative language. It should be noted that the features mentioned here will only work correctly if the source file is in the correct major mode.

The action taken by the *Tab Key* is redefined. Tab stops are set every four spaces. Zmacs will tabify backwards down the line so that the line is not full of a mixture of tabs and spaces. As normal in Zmacs, to get a real Tab character you should type *C-Q Tab*.

"Edit function definition" options such as *M-*, and the edit option in the debugger have been modified so that this will work correctly for L100 source code. Two definitions for functions will be found. One will be in the generated code and one will be in the source code. Thus *M-*, with a numeric argument will find the required definition.

F.1 Commands Supported within Zmacs

The following commands are supported in such a manner that they will work in an appropriate fashion for any buffer or file, which is of any language derived from L100. Wherever reference is made to the L100 compiler the user should interpret this as meaning the L100 compiler or the L100 derived compiler for that language.

- | | |
|-------------------------------|--|
| L100 Mode | This command will set the major mode of the current buffer to be <i>L100 Mode</i> . Analogous commands are defined for any L100 derived languages. |
| Compile Region | This command will cause the current region to be compiled through the L100 and then through the Common Lisp compiler. If a region is marked then this will be compiled. If no region is marked then Zmacs will compile the current section. Sections are delimited by top level definitions starting in the first column of a line. This definition will be compiled. When the L100 compiler is parsing the source code it prints it out in Zmacs's typeout window as it goes. This helps the user to find syntax errors. After this has finished the Common Lisp compiler will take over. |
| Parse Region | This command is just like <i>Compile Region</i> only the code is not passed on to the Common Lisp compiler. This command, therefore, is useful as a syntax check. |
| Parse and Print Region | This command is like <i>Parse Region</i> only after the L100 compiler has finished parsing the defined region the generated code is printed out in the Zmacs typeout window. This command is intended mostly as a debugging aid for compiler developers. |
| Parse and Print Buffer | This command is like <i>Parse Buffer</i> only after the L100 compiler has finished parsing the buffer the generated code is printed out in the Zmacs typeout window. This command is intended mostly as a debugging aid for compiler developers. |

Parse into Buffer	This command is like <i>Parse and Print Region</i> only instead of displaying the generated code from the compilation in the Zmacs typeout window it is ground into a, possibly new, buffer which the user specifies.
Compile Buffer	This command is just like <i>Compile Region</i> only the region selected is the whole buffer.
Parse Buffer	This command is just like <i>Parse Region</i> only the region selected is the whole buffer.
Compile File	This command is just like the normal <i>Compile File</i> command except that it will, when given a file to compile, which has as its mode the name of an L100 derived language, compile it through the relevant compiler before compiling it through the Common Lisp compiler.
Parse File	This command is much like <i>Compile File</i> only the file must be one which has as its mode the name of one of the L100 derived languages. The file is parsed but is not put through the Common Lisp compiler.
Compile and Load File	This command acts just like the system's <i>Compile and Load File</i> command unless the file in question has the name of an L100 derived language as its mode. In this case the file is parsed, the resulting code is compiled by the Common Lisp compiler and then the binary file is loaded.
Compile Changed Sections	This command and all of its related commands, such as <i>Tags Compile Changed Definitions</i> , are supported in the appropriate manner. Changed definitions in a buffer, whose mode is <i>L100 Mode</i> or one of the L100 derived major modes, will be compiled through the relevant compiler and will then be compiled and loaded by the Common Lisp compiler.

F.2 Keyboard Commands Supported within Zmacs

Most Zmacs keyboard commands act as they do in Lisp based modes. The following L100 supported commands are of significance.

C-Sh-c	This command has the same effect as <i>Compile Region</i> .
Mouse-1-R Compile Region	This command has the same effect as <i>Compile Region</i> .
C-Sh-a	This command has a similar effect to that which it has in Lisp based major modes. In L100 based modes the user must have the cursor somewhere within the name of the function in question, as opposed to somewhere within a call to a function. This is because L100 based languages support used defined syntax and function call types. It is therefore not obvious which function the user might mean in the general case.
C-Sh-d	This command works in a similar manner to <i>C-Sh-a</i> , except that it prints the documentation string of the function in question, as opposed to the arglist.

G. The Poligon Zmacs Interface

Like any language based on L100, Poligon is interfaced to *Zmacs*. There is a Poligon major mode, which should be specified in the *Mode Line* of all Poligon source files. In other respects the Poligon major mode acts just like the L100 major mode with one major exception. This is that the Poligon major mode saves context between compilations. This is done so as to support the incremental compilation of Poligon code, which needs the context associated with the *Class Declarations* in order to generate the correct code. For this reason the first code compiled in the Poligon *Compiler* within *Zmacs* must be the *Class Declarations* (see Section 3.4). Failure to do this will result in an error stating that the *Class Declarations* have not been compiled. From this point on any *Knowledge Source*, for instance, can be compiled from within *Zmacs*.

H. The L100 Defsystem Interface

The L100 language, and any languages derived from it, are interfaced to the *Defsystem* utility. This allows the compilation of non-lisp files in the appropriate manner, taking into account their compilation and load dependencies.

Files in and L100-like language can be included in modules in *Defsystem* declarations. For instance you can do any of the following.

```
(:module a-module ("host:directory;something.L100"))
```

or

```
(:module a-module  
  ( ("host:directory;something.L100"  
    "host:directory.another-directory;something.lisp")) )
```

or

```
(:module a-module  
  ( ("host:directory;something.L100"  
    "host:directory.another-directory;something.lisp"  
    "host:directory.yet.another-directory;something")) )
```

Clearly, unlike normal Lisp compilations, you can provide a pathname for the intermediate Lisp file.

To support these changes a number of new general purpose transformations have been implemented. All of these transformations use the source files' mode lines to find the right compiler, including Lisp where appropriate. These are as follows.

- :General-Parse** Parse the files in the module.
- :General-Parse-Init** Parse the files in the module looking after the specified dependencies like *Compile-Load-Init*.
- :General-Compile** (Possibly) Parse and Compile the files in the module.
- :General-Compile-Load** The *Compile-Load* version of *:General-Compile*.
- :General-Compile-Load-Init** The *:Compile-Load-Init* version of *:General-Compile*.
- :Load-Into-Zmacs** Load the files in the module into Zmacs.

For each L100-like language there is a set of transformations specific to that language. These do not inspect the mode lines in order to find the language for compilation and force compilation in the language specified. The transformations are analogous to the ones mentioned above. For a language called "Foo" they would be as follows.

- **:Foo-Parse**
- **:Foo-Parse-Init**
- **:Foo-Compile**
- **:Foo-Compile-Load**
- **:Foo-Compile-Load-Init**

The package default option in Defsystem will cause the intermediate lisp files to be generated with the new package specified in the mode line. Thus if a file is in L100 mode and in the *TV* package it can be compiled for the *Foo* language and into the *Foo* package by using the *:Foo-Compile* transformation and the package override facility as is shown below.

```
(defsystem an-example
  (:Name "An example system showing some new
        transformations.")
  (:Package Foo)
  (:Pathname-Default "Host:Directory;"
  (:Module module-1
    (("Host-2:Directory-2;test-1.L100" "test-1")))
  (:Module module-2
    (("Host-2:Directory-2;test-2.L100"
      "Host-2:Directory-2;test-2.lisp"
      test-2")))
  (:Foo-Compile-Load module-1)
  (:Foo-Compile-Load-Init
    module-2 (module-1) (:Fasload module-1)))
```

I. Listeners for L100 based languages

For each language based on L100 there is a flavor of listener window like a Lisp listener only which evaluates expressions in that language and compiles definitions for that language. The flavors are defined by appending "-Listener" to the name of the language and they are defined in the TV package. Thus there is a flavor of listener called *tv:L100-Listener*. Such listeners, by default, evaluate expressions and print their values. If the user wants to do such things as define functions then the text must be preceded by a colon. The value of a definition is always nil. A blank line is used to denote the end of input.

An example session typing to such a listener might be as follows :-

```
[L100]> 2 + 2 ;;; Evaluate the expression "2+2"4
[L100]> define a-function(x)
      ;;; Define a function. This is compiled.
      print(x)
      enddefine
nil
[L100]>
```

J. The Poligon language grammar

Below is a formal definition of the grammar of Poligon. The textual representation of the grammar has had the attributes associated with the code generation removed in order to enhance its legibility but is in all other respects identical to the real grammar.

Poligon's *Grammar* is pretty simple. The *Grammar* is in a sort of *BNF* form. Upper case words denote keywords. *:Keywords* are used to denote the names of productions and the names of non-language basic symbols. The latter include *:identifier*, *:number*, *:bra* and *:ket* (open and close parenthesis) and *:lsq* and *:rsq* (open and close brackets). Optional items are enclosed in brackets. Alternative items are enclosed in braces and are separated by "v" signs.

It should be noted that this *Grammar* has been automatically processed from the real *Grammar* used by the system. Thus decisions concerning how the *Grammar* is distributed about the productions were made on the basis of ease of code generation and not of legibility. Equally, some features might appear to be superfluous, for instance the definition of *:elsepart* and *:nelsepart*, which are the same. The code generation components for these two are distinct and so they are distinguished in the *Grammar*. It is, however, hoped that the user will not have to resort to a close scrutiny of the *Grammar* in order to make use of the language. The starting production for any attempted compilation in Poligon is called *:Poligon*.

```
:lexpr ::= λ :routinebody ENDλ
:actionitem ::= :proposition v :execution
:actionitems ::= :actionitem [ :actionitems ]
:actionpart ::= ACTION PART :colon :definitionpart :optforcing
               :actionitems :ruleotherwisepart :timeoutpart
:activation ::= ACTIVE :colon :expr
:actualparamlist ::= :bra [ :actualparams ] :ket
:actualparams ::= :expr [ :commaexprs ]
:andbinditemlist ::= &BIND :binditems
:andextendedopexpr ::= '& :operator :identifier [ :opat ]
                    [ :andextendedopexpr ]
:andletitemlist ::= &LET :letitems
:andstringlist ::= '& :restofstringlist
:appendeditem ::= '↓ v '?' :simple
:arguments ::= ARGUMENTS :colon :params
:bindecl ::= BIND :binditems IN :statements ENDBIND
:binditems ::= :letitem [ :andbinditemlist ]
:brackettedexpr ::= :bra :expr :ket
:casedecl ::= CASE :expr OF :choiceitems [ :otherwisepart ] ENDCASE
:change ::= CHANGE TYPE :colon :update v :event v :expect
               v :linkchange v :linksubsystem
               v :unlinkchange
               v :unlinksubsystem
               v :recycle v :discard
:choiceitems ::= CHOICE :actualparams :colon :expr [ :choiceitems ]
:classdecls ::= CLASS :identifier :colon :metaclasses :superclasses
               :poligonfields :printas [ :classdecls ]
:classdefinitions ::= CLASS DEFINITIONS FOR MODEL :expr
                   :colon :classdecls
:classupdate ::= UPDATED CLASS :fieldsorslotupdates
:colonemptyorexprs ::= :colon :emptyorexprs
:colontoplevelstatemnt ::= :colon :toplevelstatement
```



```

:commaexprs ::= :comma :actualparams
:commaforcedidentifiers ::= :comma :forcedidentifiers
:commaformals ::= :comma :formalparams
:commaparams ::= :comma :params
:conditionpart ::= CONDITION PART :colon :optforcing WHEN :colon
                    :expr :optforcing IF :colon :expr
:constdecl ::= CONSTANT :identifier '≡ v '== :expr
:definitionbody ::= DEFINITIONS :colon :definitions
:definitionpart ::= [ :definitionbody ]
:definitions ::= :multivalues [ :definitions ]
:defstatement ::= DEFINE :identifier :routinebody ENDDFINE
:deletion ::= DELETE :colon :expr
:discard ::= DISCARD UPDATED NODE :colon :expr
:docstring ::= DOCUMENTATION :stringlist
:dostatement ::= DO :expr
:eager ::= EAGER :colon :forcedidentifiers
:elseifparts ::= ELSEIF :expr THEN [ :statements ] [ :elseifparts ]
:elsepart ::= ELSE :statements
:elseunlessparts ::= ELSEUNLESS :expr DO [ :statements ] [
:elseunlessparts ]
:emptyorexpr ::= EMPTY v :expr
:emptyorexprs ::= EMPTY v :exprs
:evaluateditem ::= '↑ v '^ :simple
:event ::= CAUSE EVENTS UPDATED NODE :colon :expr UPDATED :fieldsorslots
                    :colon :identifiers
:execution ::= EXECUTE :colon :statements
:exestatement ::= EXECUTE :expr
:expect ::= EXPECT RULE :colon :identifier NODE :colon :expr
                    :fieldsorslot :colon :identifier :expectbody
:expectbody ::= :expectwhen :expectif :activation
                    :deletion :timeout :definitionpart
:expectif ::= IF :colon :expr
:expectwhen ::= WHEN :colon :expr
:expr ::= :simplewithparts [ :opexpr ]
:exprs ::= :actualparams
:extendedopexpr ::= '& :operator :identifier [ :opat ]
                    [ :andextendedopexpr ]
:fieldnames ::= :identifier :optdefault [ :fieldnames ]
:fieldsorslot ::= FIELD v SLOT
:fields ::= :fieldsorslots :colon [ :fieldnames ]
:fieldsorslots ::= FIELDS v SLOTS
:fieldsorslotupdates ::= :fieldsorslots :colon :slotdefinitions
:forcedidentifiers ::= :identifier [ :commaforcedidentifiers ]
:forcing ::= FORCE :colon :forcedidentifiers
:formalparam ::= :identifier [ :types ]
:formalparams ::= :formalparam [ :commaformals ]
:forwhich ::= FOR WHICH ELEMENT :operator :identifier :operator :expr
:identifiers ::= :identifier [ :identifiers ]
:ifexpr ::= IF :expr THEN [ :statements ] [ :elseifparts ] [ :elsepart ]
                    ENDIF
:initialisation ::= :inits :colon :statements
:inits ::= INITIALISATION v INITIALIZATION
:knowledge source ::= KNOWLEDGE SOURCE :colon :identifier
                    :definitionpart [ :rules ]
:lambdaexpr ::= LAMBDA :routinebody ENDLAMBDA
:letdecl ::= LET :letitems IN :statements ENDLET
:letitem ::= :identifier '≡ v '== :expr
:letitems ::= :letitem [ :andletitemlist ]

```

```

:linkchange ::= LINK :expr TO :expr LINK :colon :expr CLASS
              :colon :expr UPDATED :fieldsorslots
              :colon :slotdefinitions
:linksubsystem ::= LINK SUBSYSTEM :expr TO :expr
:list ::= :LSQ [ :listvalues ] :RSQ
:listvalue ::= :appendeditem v :normallistitems
:listvalues ::= :listvalue [ :listvalues ]
:metaclasses ::= METACLASSES :colon :params
:multiletdecl ::= MULTIPLELET :multivalues IN :statements ENDMULTIPLELET
:multivalues ::= :params '= v '== :expr
:namecolonvalues ::= :identifier :colon :emptyorexpr NOEVENT
                  [ :comma ] [ :namecolonvalues ]
:nelsepart ::= ELSE :statements
:newinstance ::= NEW INSTANCE OF :expr :unlesscreate
               :classupdate :supersystems
               :slotinitialisations
:normallistitem ::= :evaluateditem v :value v :percenteditem
:normallistitems ::= :normallistitem [ :normallistitems ]
:opat ::= AT :simple
:opexpr ::= :simpleopexpr v :extendedopexpr :opsideeffects
:opsideeffects ::= :colon :unlesscreate :updatedfields
:optdefault ::= :colonemptyorexprs
:optforcing ::= :eager :forcing
:otherwisepart ::= OTHERWISE :colon :expr
:parallelisation ::= IN PARALLEL FOR EACH :identifier IN :expr
                  :definitionpart :optforcing :change
:parallelletdecl ::= PARALLELLET :letitems WHEN :expr IN
                  :statements ENDPARALLELLET
:paramlist ::= :bra [ :formalparams ] :ket
:params ::= :identifier [ :commaparams ]
:percenteditem ::= '%' :actualparams '%'
:poligon ::= :Start_of_File :tooplevelstatements :End_of_File
:poligonfieldnames ::= :identifier :optdefault [ :slotspecifiers ]
                  [ :poligonfieldnames ]
:poligonfields ::= :fieldsorslots :colon [ :poligonfieldnames ]
:poligonletdecl ::= LET :letitems IN :statements ENDLET
:printas ::= DISPLAY AS :colon :actualparams
:procedureparts ::= :actualparamlist [ :procedureparts ]
:proposeitem ::= :parallelisation v :change
:proposeitems ::= :proposeitem [ :proposeitems ]
:proposition ::= CHANGES :colon :definitionpart :proposeitems
:recycle ::= RECYCLE UPDATED NODE :colon :expr
:restofstringlist ::= :string [ :andstringlist ]
:routinebody ::= [ :paramlist ] :docstring [ :statements ]
:rule ::= RULE :colon :identifier :arguments v :ruleheader
         :definitionpart :conditionpart :selectactionpart v :actionpart
:ruleheader ::= CLASS :colon :identifier :fieldorslot :colon :identifier
:ruleotherwisepart ::= OTHERWISE PART :colon :definitionpart
                   :optforcing :actionitems
:rules ::= :rule [ :rules ]
:selectactionpart ::= :optforcing SELECT :colon :expr ACTION PART
                    :colon :definitionpart :optforcing
                    :taggedcomponents :ruleotherwisepart
                    :timeoutpart
:semistatements ::= 'Ø v ' ! :statements
:simple ::= :value v :poligonletdecl v :ifexpr v :brackettedexpr
           v :unlessexpr v :lexpr v :lambdaexpr v :multiletdecl
           v :caseddecl v :binddecl

```

```

:simpleopexpr ::= :operator :expr [ :opat ]
:simplewithparts ::= :simple [ :procedureparts ]
:slotdefinition ::= :identifier :operator NOEVENT
                  :slotdefinitionlist v :emptyorexprs
:slotdefinitionlist ::= '& :expr
:slotdefinitions ::= :slotdefinition [ :slotdefinitions ]
:slotinitialisations ::= :inits :colon :slotdefinitions
:slotspecifiers ::= :specifierkeyword :colon :expr [ :slotspecifiers ]
:specifierkeyword ::= INSERTIF v REMOVEIF v MODIFYWITH
                   v INDEXEDBY v SORTEDBY v KEYEDBY
:statements ::= :expr [ :semistatements ]
:stringlist ::= :string [ :andstringlist ]
:structure ::= STRUCTURE :identifier FIELDS :params
:subset ::= SUBSET OF :expr :forwhich v :whichsatisfies
:superclasses ::= SUPERCLASSES :colon :params
:supersystems ::= SUBSYSTEM OF :colon :exprs
:tagedcomponents ::= :expr :colon :actionitems [ :tagedcomponents ]
:timeout ::= TIMEOUT :colon :expr
:timeoutpart ::= TIMEOUT PART :colon :definitionpart :optforcing
               :actionitems
:toplevelexpr ::= :Start_of_File :expr v :colontoplevelstatememnt
                 :End_of_file
:toplevelstatement ::= :defstatement v :vardecl v :constdecl
                     v :knowledgesource
                     v :classdefinitions
                     v :structure v :initialisation
                     v :dostatement v :exestatement
:toplevelstatements ::= :toplevelstatement [ :toplevelstatements ]
:types ::= :colon :identifier [ :types ]:unlesscreate ::= UNLESS
          :colon :expr
:unlessexpr ::= UNLESS :expr DO [ :statements ] [ :elseunlessparts ]
              [ :nelsepart ] ENDUNLESS
:unlinkchange ::= UNLINK :expr FROM :expr LINK :colon :expr
:unlinksubsystem ::= UNLINK SUBSYSTEM :expr FROM :expr
:update ::= UPDATE UPDATED NODE :colon :expr :updatedfields
:updatedfields ::= UPDATED :fieldsorslotupdates
:value ::= :Identifier v :Keyword v :number v :Quoteditem
          v :Stringlist v :List v :Function_value
          v :newinstance v :subset v :parallelletdecl
:value ::= :Identifier v :Keyword v :number v :Quoteditem
          v :Stringlist v :List v :Function_value
:vardecl ::= VARIABLE :identifier '← v '← :expr
:whichsatisfies ::= WHICH SATISFIES v FAILS :expr

```

K. Poligon Language Keywords

Table K—1 shows all of the Poligon language keywords. These words are reserved and cannot be used as identifiers in user programs.

λ	Eager	If	RemoveIf
&Bind	Element	In	Rule
&Let	Else	IndexedBy	Satisfies
Action	ElseIf	Initialisation	Select
Active	ElseUnless	Initialization	Slot
Arguments	Empty	InsertIf	Slots
As	End λ	Instance	SortedBy
At	EndBind	KeyedBy	Source
Bind	EndCase	Knowledge	Structure
Case	EndDefine	Lambda	Subset
Cause	EndIf	Let	Subsystem
Change	EndLambda	Link	Superclasses
Changes	EndLet	Metaclasses	Then
Choice	EndMultipleLet	Model	Timeout
Class	EndParallelLet	ModifyWith	To
Condition	EndUnless	MultipleLet	Type
Constant	Events	New	Unless
Define	Execute	Node	Unlink
Definitions	Expect	NoEvent	Update
Delete	Fails	Of	Updated
Discard	Field	Otherwise	Variable
Display	Fields	Parallel	When
Do	For	ParallelLet	Which
Documentation	Force	Part	
Each	From	Recycle	

Table K—1 Poligon Language Keywords

L. Poligon Language Operators

Tables L—1 and L—2 shows all of the operator symbols predefined in the Poligon language and the names of the operations or functions, which they denote.

Operation	Symbol
*	*
+	+
-	-
<	<
<=	<=
<=	≤
>	>
>=	>=
>=	≥
All-Elements	@@
All-Elements	⊕
An-Element	@
An-Element	•
An-Element-Or-Nil	@?
An-Element-Or-Nil	•?
And	And
Append	↔
Append	<>
Are-The-Same	=
At-Type	As-Type
Cardinality-Exceeds	Cardinality-Exceeds
Collection-Difference	Collection-Difference
Collection-Intersection	Collection-Intersection
Collection-Union	Collection-Union
Cons	⊃

Table L—1 Poligon Language Operators

Operation	Symbol
Include-Comment	Comment
Is-Empty	Is-Empty
Is-Empty-Or-Undefined	Is-Empty-Or-Undefined
Is-In	Is-In
Is-Not-Empty	Is-Not-Empty
Is-Not-Empty-Or-Undefined	Is-Not-Empty-Or-Undefined
Is-Not-In	Is-Not-In
Is-Not-Undefined	Is-Not-Undefined
Is-Of-Type	Is-Of-Type
Is-Strictly-Of-Type	Is-Strictly-Of-Type
Is-Undefined	Is-Undefined
Modify	←
Modify	< -
Not-Equal	%=
Not-Equal	≠
Or	Or
Quotient	/
Remove-Element	< - *
Remove-Element	← *
Replace-All-Elements	← ---
Replace-All-Elements	< ---
Self	->
Self	→
With	With
Without	Without

Table L—2 Polygon Language Operators

M. Miscellaneous System Defined Functions, Operators, Macros and Variables

The following functions, macros and variables are defined in Polygon and are of general use.

Apply-Function(function, &Rest arguments) -> Applies the function *function* with arguments *arguments* as if it was called using *Apply*.

Breakpoint(&Optional (value-to-return :Null-Value-From-Breakpoint), (break-p t), (format-string "Polygon Breakpoint"), &Rest format-args) -> Executes a breakpoint when it is called, if *Break-p* delivers non-nil. If *Break-p* is a function then this function is called with *Value-To-Return* as its argument. The value of a call to this function is *Value-To-Return*, so a breakpoint can be wrapped around any expression and still be an identity operation.

Call-Function(function, &Rest arguments) -> Calls the function *function* with arguments *arguments* as if it was called using *Funcall*.

expression Comment comment-expression The use of this operator causes explanatory text to be embedded in the generated code. The function associated with the operator is called *Include-Comment* and is expected to be an Identity function on its first argument. Its second argument is the comment itself, which can be any expression, but which is typically a string. The default "Include-Comment" function throws away its second argument, but the user can define it to do anything that he wants with the comment. This form of comment is legal anywhere that an expression is legal in the Polygon language.

Coerce-To-Object(something) -> Removes all Polygon defined data structures until a basic object, for instance a flavor instance, is returned. This is useful in print methods where the true detail of Polygon's representation might be too complex. This function is particularly useful, since *Coerce-To-Object*(#<Remote Sheep-42>) is the instance *Sheep-42*.

Current-Time() -> The time at the time that user clock last ticked before this function is called in user time units, expressed as an integer.

Dont-Optimise() A macro, which switches off any compiler optimizations that the Polygon system may use. This macro can only be placed at the head of a function definition or just after the *In* of a *Let* or *MultipleLet* construct. This prevents certain functions, like *Is-In* and *Field Selection Operations* from being open coded. This should result in the model being more intelligible when debugging, but it will probably be significantly slower.

Floated-Current-Time() -> The time that this function is called in user time units, expressed as a floating point number.

Halt-Polygon() Arrests the process in which the Polygon simulation runs.

Has-Type(something, type) This macro declares that *Something* is of type *Type*. For instance, *Has-Type(nodes, list)* declares that

nodes is always of type *List*. This information can be used by the compiler in its optimizations. For information on this topic please see Appendix Q.

Is-Not-Nil(something) Is true if *Something* is not *Nil*.

Optimise()

A macro, which maximizes the compiler optimizations performed on the function in which it is used. This macro can only be placed at the head of a function definition or just after the *In* of a *Let* or *MultipleLet* construct. A significant feature of this is that *Tail Recursion Optimization* is turned on. This is the way in which the user would implement a *Loop* in Poligon.

Run-Poligon(&Optional (Parallel-P nil)) Initializes the Poligon system so that it will start a simulation as soon as the menu option is selected. If *Parallel-P* is true then the system will be initialized in such a way as to prepare is to run in *Parallel* mode. This function must be called in order to toggle between the *Serial* and *Parallel* modes.

Selected-Circuit This variable has as its value the name of the currently selected CARE circuit or nil (default). Setting this variable before the model is loaded will allow the user to select any of the circuits which he has loaded. If this is not done then the system will select the most recently loaded circuit for any simulations which are performed in the *Parallel* mode.

Returns-Coerced-Result(Function-Name) Declares that *Function-Name* returns a result, which is guaranteed not to be a *Future* or a *Multiple-Values* object.

Stop-Poligon(&Optional (No-Statistics nil)) Stops the Poligon simulation by aborting it and then killing the process in which it was running. By default a statistics file will be printed out, if it can be. To prevent this make *No-Statistics* true.

Unhalt-Poligon() Unarrests the process in which the Poligon simulation runs.

With-Printing-Bindings(&Body Body) This macro executes *Body* with useful bindings for the system print control variables. This makes sure that anything printed inside *Body* comes out neatly.

Without-Poligon-Clock(&Body Body) This macro executes *Body* with the simulation clock stopped.

N. Functions for processing Multiple Values

The following functions create or manipulate *Multiple Values*.

Multiple-Values(&Rest values) -> a multiple values object, which contains the values *values*.

First-Of-Multiple-Values(values) -> the first value of the multiple values represented by *values*.

Second-Of-Multiple-Values(values) -> the second value of the multiple values represented by *values*.

Multiple-Values-Element-Of(n, values) -> the *nth* (zero indexed) value from the multiple values object represented by *values*.

All-Values-Of(values) -> the list of all of the values represented by *values*.

Is-A-Multiple-Values(something) -> non-nil if *something* is a multiple values object.

O. Output Routines

The routines provided by Polygon for output are as follows.

Debug-Format(control-string, &Rest format-args) Formats something into the *Lisp pane*.
Debug-Heading(control-string, &Rest Heading-args) Formats something into the *Lisp pane* and underlines it.
Debug-Princ(something) Prints something into the *Lisp pane*.
Debug-Print(something) Prints something into the *Lisp pane*.
Debug-Terpri() Throws a newline in the *The Lisp pane*.
Polygon-Format(control-string, &Rest format-args) Formats something into the *Lisp pane*.
Polygon-Princ(something) Prints something into the *Lisp pane*.
Polygon-Print(something) Prints something into the *Lisp pane*.
Polygon-Terpri() Throws a newline in the *Lisp pane*.
Polygon-Format-and-wait(control-string, &Rest format-args) Formats something into the *Lisp pane*.
Polygon-Princ-and-wait(something) Prints something into the *Lisp pane*.
Polygon-Print-and-wait(something) Prints something into the *Lisp pane*.
Polygon-Terpri-and-wait() Throws a newline in the *Lisp pane*.
Trace-Format(control-string, &Rest format-args) Formats something into the *Lisp pane* in the standard trace font.

These routines have the following effects. The *Debug-** routines are designed for debug purposes only. They stop the clock and use internal mechanisms which attempt to ensure that the debug printing activity will not affect the simulation. The *Polygon-Format / Print / Princ / Terpri* procedures are true user output procedures. They cause the desired output to be routed through a single channel in the Polygon system, which is associated with the *Polygon-Blackboard* Node. Output of this type counts as part of the user's program and is subject to the normal constraints of the simulation of the version on which he is working. This set of output procedures does not cause the system to wait whilst output is being done. Because of this there is no guarantee that output will come out in the order expected by the user. If synchronized output is needed then the next set of procedures should be used. The *Polygon-Format / Print / Princ / Terpri-and-wait* procedures have the same effect as *Polygon-Format / Print / Princ / Terpri* except that they wait until the output has finished. This means that output being done in a given block of code will come out in the correct order. The synchronization associated with waiting for output in a predictable order is potentially expensive so the *Polygon-Format / Print / Princ / Terpri* procedures should be used wherever it is meaningful.

P. User Defined Parallelism

It is generally the case that the control of parallelism in Poligon models is left to the system. There are cases, however, when it may be reasonable, because of the size of a piece of computation to attempt to do parts of it in parallel. To this end Poligon supports a mechanism for achieving parallelism at the expression level by a function call model, as opposed to the strongly co-routine modelled parallelism available in Poligon. This is done by the use of a *ParallelLet* construct, an example of which follows.

```
Define Fibonacci (n)
  ParallelLet n-1 ≡ Fibonacci (n - 1)
             &Let n-2 ≡ Fibonacci (n - 2)
    When n > 10
    In n-1 + n-2
  EndParallelLet
EndDefine
```

The above is a trivial definition of a function for the determination of the *n*th Fibonacci number. When *n* is greater than 10 it is considered worth doing the calculation in parallel and it will therefore be done as such, otherwise the values will be calculated in series. It should be noted that the values denoted by the names *n-1* and *n-2* are *Futures*. They will thus only be defutured when they meet "+", a strict operator.

This construct is very closely related to *QLet* in QLisp. The determination of when operations should be performed in parallel is left entirely to the user.

Q. Hints on Programming Efficiency in Poligon

In Poligon it is possible, as in any system, to write logically correct but, nevertheless, hopelessly inefficient *Models*. The purpose of this section is to give some hints on how a model should be organized in order to maximize its efficiency, without compromising its intelligibility or its semantics. It should be noted that a number of the suggestions below are not only likely to lead to efficient *Models* but also to more reliable and consistent *Models*.

Q.1 Reading from Fields

All references to the *Fields* of a particular Node which are referred to within a rule or function should, wherever possible and meaningful, use the *a-node* & *field-1* & *field-2* mechanism to read all of the values in one go. This is not only more efficient but guarantees that all of the values are read consistently.

Significant performance improvements can be achieved by optimizing *Fields*, which act like dictionaries, by telling the system that the elements in the dictionary will be indexed by an integer. For more information on this please see Section 3.3.4.3.

If *Fields* are to be read in the body of a rule from the Node, which has had the rule triggered on it then, if possible, these should be read during the evaluation of the *When Part*. If there is no need to read any of these values in the *When Part* then the reading of them should be *Forced* before the *When Part* is evaluated.

When using the function *Map-Over-A-Collection* on a *Collection* of *Nodes* it is advisable, when the function being mapped refers to the node it is passed as an argument, to read any *Fields* at the top of the function. This allows the system to spot these read operations before doing any potentially expensive work so that it can distribute the function invocations. This is particularly significant for unordered *Collections*, for which the gain is likely to be greatest.

Q.2 Efficiency from Typing

The efficiency of some functions and operators can be increased by type declarations using the *Has-Type* macro or the *The* special form. Notable amongst these are *=*, *≠* and *Is-In*.

Some efficiency can be gained by declaring all functions which return coerced results to have such, using the function *Returns-Coerced-Result*. This is worth doing even if the result type of the function is not known precisely.

Q.3 Efficiency in the Action Parts of Rules

If the expression on the right hand side of an *Update Operator* makes reference to the old value of the *Field* being updated, either directly or indirectly through definitions, then it is worth defining a *Field* modification behavior specifier function (see Sections 3.3.4 and 3.4.2.1). This has the benefits that you do not need to read the *Field* before the update is made, which is more efficient, and there is no possibility of another rule changing the value of the *Field* between the time that the action parts are evaluated and the time that the update occurs.

When the action part of a rule fires it attempts to find good places to execute the code. This is usually on the Node to which the update is to be made. A problem can occur here if some definitions have not yet been evaluated by the time that this remote update is made, a deadlock could occur because the context in which the rule is being fired is waiting for the successful completion of the remote update and the Node being updated may have to communicate with the *Context* in order to evaluate the definition. To circumvent this the system spawns a new context at the updated node to execute the update. This is referred to as *Punting* and is much less efficient than the non-*Punting* case. To help the programmer to eliminate this inefficient behavior by *Forcing* definitions that need to be *Forced* before the update is performed *Trace & Break* options have been provided. If it is meaningful to do so it can be more efficient to *Force* all definitions that will be needed in the *Action Part* of a rule at the beginning of the *Action Part*.

Q.4 Miscellaneous Hints on Efficiency

Whenever a *Field* of a Node is read more than once in the body of a rule or function it should be read only once and should have its value named using a *Definition*, *Let* or *MultipleLet* form.

Care in defining appropriate equality testing predicates for structures can result in significant improvements in performance. This applies also to the *:No* specifier. See Section 3.3.2.3.

If the user is not concerned with complex initialization and cache building matters for a particular class of *Nodes* it may be advisable to use a quoted class name argument to the *New Instance Of* construct. See Section 3.3.3.3.

Whenever a number of rules, which trigger on the same *Field* of a Node have preconditions (*If Parts*), which are mutually exclusively non-nil, these rules should be gathered together into a case-like rule with a *Select Part*. Since only one rule can possibly fire this reduces the amount to needless replication of effort and of invocation of rule activation contexts.

When the model has a need to determine whether there are any matches for a given condition it is much more efficient to do a *Subset* operation and then test the value of this operation with the functions *Cardinality-Exceeds*, *Collection-Is-Empty* or *Collection-Is-Not-Empty* than it is to test the value of the *Subset* operation to see whether its *Number-Of-Elements* = 0. This is because the former operations need only wait for one affirmative reply from the *Subset* operation, whilst the latter must wait for all replies.

R. Functionality that is in place, but does not work

R.1 Run-Time Property Inheritance

For a number of reasons, the features originally provided for run-time property inheritance in Poligon have died from software rot, though the code to support them is still in place. These facilities are described in this section.

R.2 Property Inheritance

Property Inheritance in Poligon comes in three main forms. These are:

- Compile-time
- Run-time, through *Supersystems*
- Run-time, through *Links*

The compile-time form is completely different from the two run-time forms so they will be described separately.

R.2.1 Compile-time Property Inheritance

Compile-time Property Inheritance takes the form of the inheritance of *Fields* by means of the class hierarchy. This form of property inheritance has always worked in Poligon.

Classes inherit *Fields* on behalf of their *Instances* so that when an *Instance* of a class is made at run-time it will have within itself all of the *Fields* specified for both that class and all of its *Superclasses*. Thus, different *Instances* of the same class will have separate copies of fields of the same name. It is by this means that each *Sheep* in the example in Section 3.4 can have a different colour. Thus this form of inheritance is the inheritance of *characteristics*, not the inheritance of *values*. Because a Node of any given class has a physical copy of each of the *Fields* specified in its *Superclasses* it is possible to trigger rules on these *Fields* just as one would on a *Field* declared for the specific class in question.

R.2.2 Run-time Property Inheritance

Run-time Property Inheritance comes in two forms, which are similar in many senses. These are inheritance through *Supersystems* and inheritance through *Links*. These two forms of inheritance are distinct from the compile-time form because they can be thought of as being the inheritance of the operations of *Fields* rather than characteristics of those *Fields* themselves. Thus, it is possible to inherit the *value* of a *Field* at run-time, without inheriting a copy of the *Field*. Similarly it is possible to inherit the ability to write, to cause an *Event* on or make an *Expectation* on a non-local *Field*.

When a reference is made to a non-local *Field* the system attempts to inherit it from somewhere else. It first tries to inherit it from the supersystems of the Node in question. Failing this it tries any *Links* that might be defined. Failing this an error is signalled.

R.2.2.1 Property Inheritance through Supersystems

When a reference is made to a non-local *Field* the system attempts to inherit first from the *Supersystems* of the Node in question. It does this recursively, depth-first starting at the

front of the values of the *Supersystems Field*. If it finds the *Field* that it is looking for it does the operation there and then returns. If it fails to be able to inherit the *Field* it returns in such a manner as the system will then try to inherit from any *Links* that might be present.

An example of the use of this form of inheritance might be that a *Sheep*, from the example in Section 3.4 might want to know about the location of the *Flock* of which it is a member. This is a characteristic of the *Flock* as a whole, not of an individual *Sheep* but could be accessed simply by the expression:

a-sheep•Flock-Location

Where *a-sheep* is a *Sheep* and *Flock-Location* is the name of a *Field* in the *Flock* of which *a-sheep* is a *Subsystem*.

R.2.2.2 Property Inheritance through Links

When the system fails to inherit a field from any of the *Supersystems* of a particular Node it tries to inherit it from the *Nodes* to which the Node has been linked by the *Link* mechanism. This is described fully in Section R.2.4.1, the section concerning the *Link Type of Change*. The manner in which the inheritance happens is just like that for inheritance through *Supersystems*, it happens depth-first, starting from the first element in the *Latest* value of the *Links* field for the Node in question. The purpose of the *Link* mechanism is to provide a means of expressing relationships other than "Is an instance of" and "Is subservient to/Is a component of". Thus in the example in Section 3.4 one might express the fact that a particular sheep is in a barn by linking it to an *Instance* of the class *Barn* with a *Link* called "Is In". The sheep would then be able to inherit the characteristics of the barn.

R.2.3 System Defined Fields for Property Inheritance

The following are the system defined fields that are used to support property inheritance.

Inherit	The value of this field is always <i>Empty</i> . Its purpose is to provide a field, on which rules can be hung so that <i>Property Inheritance</i> can be monitored. When a property is inherited along a <i>Link</i> the <i>Inherit Field</i> of the Node which represents the <i>Link</i> is triggered.
Is-A-Link	The <i>Latest</i> value of this field is a flag, which is non- <i>nil</i> if the Node in question is being used as a <i>Link</i> Node. For more information on <i>Link Nodes</i> please see Section R.2.4.1.
Links	The values of the <i>Links</i> field are the <i>Links</i> associated with the Node in question.. For more information on <i>Links</i> please see Section R.2.4.1. These <i>Links</i> are represented as instances of the system internal data structure <i>Link-Cell</i> . Functions for processing <i>Links</i> can be found in Appendix S.

R.2.4 Change Types Used by the Property Inheritance Mechanism

A number of change types are implemented to support the property inheritance mechanism. These are described in this section.

Link	Allows the model to connect <i>Nodes</i> together with a bidirectional <i>Link</i> . This <i>Link</i> can have arbitrary properties associated with it and allows <i>Property Inheritance</i> of fields.
Link Subsystem	Allows the model to connect <i>Nodes</i> together with a <i>Subsystems</i> to <i>Supersystems</i> <i>Link</i> . This <i>Link</i> allows <i>Property Inheritance</i> of fields.

- Unlink** Allows the model to disconnect *Nodes*, which are already *Linked* together by a *Link*.
- Unlink Subsystem** Allows the model to disconnect *Nodes*, which are already *Linked* together by a *Subsystems* to *Supersystems* *Link*.

R.2.4.1 The Link type of Change

The *Link* type of *Change* allows the user to use the *Blackboard* as a sort of *Entity/Relationship/Attribute* database. *Links* are represented as *Nodes*. This means that they are first-class citizens in the Poligon system. A *Link* can therefore have any number of *Fields* defined for it, as well as a name, which is the mechanism by which the system distinguishes between *Links* between the same pair of *Nodes*.

A *Link* between two *Nodes* can be made in the following fashion.

```
Changes :
  Change Type : Link a-node to another-node
  Class : a-class-of-link-nodes
  Link : "The name of the Link"
  Updated Fields :
    a-field-in-the-link-node ← 42
```

This piece of code causes a *Link* to be made between *a-node* and *another-node*, which is of the class *A-Class-Of-Link-Nodes*. The link has the name "*The name of the Link*" and is having its *Field* called *A-Field-In-The-Link-Node* modified to have the value 42 added. If a *Link* already exists between the *Nodes* specified, which has the name specified then that *Link* is used, otherwise a new one is created. Any number of *Fields* in the *Link* Node can be initialized in the *Updated Fields* component.

One of the major reasons for the *Link* mechanism is to support Poligon's *Property Inheritance*. If the user attempts to read a field from a *Node*, which does not have such a field then the system will attempt to inherit the field from a *Node*, to which the *Node* in question is linked by the *Link* mechanism if it cannot find it by other means. This is described fully in Section R.2, the section concerning *Property Inheritance*. Any, possibly cyclic, graph of *Nodes* can be linked. The search for a *Node*, which has a matching field happens in depth-first fashion along the *Links*. Suitable checks are made which force backtracking in the event of an attempt at inheritance along a cyclic path.

The values, which are inherited are not cached. This is because in the general case the cost of maintaining cache consistency is likely to be as great as the cost of fetching the value again.

R.2.4.2 The Link Subsystem type of Change

The *Link Subsystem* type of *Change* allows the user to add new *Subsystems* to *Supersystems* *Links* at times after a *Node* has been created. This allows changes in the understanding of the world to be reflected in the *Subsystems* and *Supersystems* *Fields*. An example use of this construct is shown below.

```
Change Type : Link Subsystem a-node To new-supersystem
```

In this example the *Node a-node* is to have *new-supersystem* added to it as a new *Supersystem*. Similarly *new-supersystem* will have *a-node* added to its *Subsystems*.

R.2.4.3 The Unlink type of Change

The *Unlink* type of *Change* allows the user to delete *Links*. This *Change Type* is the inverse operation of *Link*. An example of the use of this construct is shown below.

Change Type : Unlink a-node From old-linked-to-node
Link : 'a-link-name

In this example the Node *a-node* is to have the *Link* called *a-link-name* unhooked between itself and *old-linked-to-node*. If no such *Link* is present then nothing happens.

R.2.4.4 The Unlink Subsystem type of Change

The *Unlink Subsystem* type of *Change* allows the user to delete existing *Subsystems/Supersystems Links*. This *Change Type* is the inverse operation of *Link Subsystem*. An example of the use of this construct is shown below.

Change Type : Unlink Subsystem a-node From old-supersystem

In this example the Node *a-node* is to have *old-supersystem* removed from its *Supersystems*. Similarly *old-supersystem* will have *a-node* removed from its *Subsystems*. If no such *Link* is present then nothing happens.

S. Functions for Processing Links

Links in the Poligon system are represented as *Nodes*. A field in each of the *Nodes*, which is linked holds a list of all of the *Links* coming out of the particular Node. The value of this list of *Links* is a list of *Link-Cell* data structures. These represent the ends of the *Links*.

The following functions are provided for the processing the *Links* of a Node.

Link-Between(a-node, another-node) -> A *Link* Node, which links *a-node* to *another-node* or *nil* if there is no such *Link*.

Link-Between-Named(a-node, another-node, link-name) -> A *Link* Node, which links *a-node* to *another-node* whose name = *link-name*, or *nil* if there is no such link.

The following functions are provided for the processing of *Link-Cells*.

Is-A-Link-Cell(a-link-cell) -> *T* if *a-link-cell* is indeed a *Link-Cell*.

Link-Cell-Link(a-link-cell) -> The *Link* Node associated with the *Link-Cell*.

Link-Cell-Link-Name(a-link-cell) -> The name of the *Link*.

Link-Cell-Linked-to(a-link-cell) -> The Node at the other end of the *Link*.

Index

- ! 8
- \$Value 36
- % 12
- %= 20
- & 9, 49
- = 20, 117
- =% 20
- Abort 59
- Abort After Input File Closed 63
- Abstract Superclass 30
- Action on Deadlock 61
- Action Part 43, 45, 66
- Active 51
- AGE 1
- AGE \$Value types
 - All 36
 - Latest 2
- AGE Updates types
 - Modify 2, 48, 78, 82
- All 36
- All-Elements 88
- All-Values-Of 111
- Allow All Trace Within Abortable Code 60
- An-Element 88
- Apply 49
- Apply-Function 90, 91, 109
- Arguments 50, 52
- As-Type 89
- Associate 22, 25, 88
- Automatic GCing 62
- Bag 18, 25, 87, 88, 90
- Bag processing functions
 - Bag 88
 - Is-a-Bag 88
- Basic-Slot 37, 50
- Begin 8
- Binding 13
- Blackboard 3
- BNF 101
- bra 101
- Break Clock Ticks 63
- Break Message Punting 64
- Break Signal Records 63
- Breakpoint 109
- C-S-a 94
- C-S-c 94
- C-S-d 94
- Call-Function 109
- Cardinality-Exceeds 89, 118
- CARE 18, 21, 23, 28, 59, 60, 62, 63, 110
- CARE Debug History 62
- CARE Evaluator Processing 62
- Cause Events 46, 47, 49
- Change 46, 121
- Change Type 46
- Change Types
 - Cause Events 46, 47, 49
 - Discard 47, 66, 121, 122
 - Expect 47
 - Link 120, 121
 - Link Subsystem 120, 121
 - Recycle 47
 - Unlink 122
 - Unlink Subsystem 121, 122
 - Update 47, 48
- Changes 45, 46, 78, 79, 82
- Circuit 59
- Class 30
- Class Declarations 7, 69, 73
- Class Definitions For Model 29
- Class Node 24, 25, 54
- Class Nodes 23, 35, 38, 58
- Clock 34
- Code_structure 85
- Coerce-To-Object 35, 109
- Collection 22, 26, 46, 87, 88, 89, 90, 91, 117
- Collection processing functions
 - All-Elements 88
 - An-Element 88
 - As-Type 89
 - Associate 88
 - Cardinality-Exceeds 89
 - Collection-Difference 89
 - Collection-Intersection 89
 - Collection-Is-Empty 89, 118
 - Collection-Is-Not-Empty 89, 118
 - Collection-Union 89
 - Copy-Collection 89
 - Current-Number-Of-Elements 27, 89
 - Element-of 89, 90, 91
 - Fold 89
 - Fold-Right 89
 - Get-Property 89
 - Head 89, 90
 - Is-A-Collection 90
 - Is-An-Ordered-Collection 90
 - Is-An-UnOrdered-Collection 90
 - Is-In 90, 117
 - Is-Not-In 90
 - Join 90
 - Make-a-Collection-of-Type 90

- Map-Over-A-Collection 90, 117
- Number-of-Elements 90, 118
- Sort-Collection 90
- Tail 90
- The-Eighth 90
- The-Excluded-Subset 90
- The-Fifth 90
- The-First 90
- The-Fourth 90
- The-Last 90
- The-Ninth 90
- The-Second 90
- The-Seventh 91
- The-Sixth 91
- The-Subset 91
- The-Tenth 91
- The-Third 29, 91
- Uniquise 91
- With 91
- Without 91
- Collection-Difference 89
- Collection-Intersection 89
- Collection-Is-Empty 89, 118
- Collection-Is-Not-Empty 89, 118
- Collection-Union 89
- Colon 15
- Colon Keyword 2, 8, 13, 15, 85, 101
- Comma 12
- Comma At-sign 12
- Command Menu Commands
 - Abort 59
 - Circuit 59
 - Configuration 59
 - Dribble Off 59
 - Dribble On 59, 62, 63
 - Log 59, 62
 - Parallel 60
 - Parameters 59, 60, 65
 - Redisplay 65
 - Run 65, 110
 - Serial 65
 - Statistics 62, 65
- Comment 109
- Comments 8, 9
- Common Lisp 1, 23, 57, 69
- Compilation Factor 61
- Compile and Load File 69, 94
- Compile Buffer 94
- Compile Changed Sections 94
- Compile File 94
- Compile Region 93, 94
- Compile-Load 97
- Compile-Load-Init 97
- Compile-time Property Inheritance 119
- Compiler 3, 57
- Condition Part 44, 45
- Configuration 59
- Cons 88, 90
- Constants 16
- Context 58, 118
- Control Frame 59
- Control Frame, components
 - Lisp pane 59, 113
 - Message Pane 59, 63, 66
 - Rule Monitor Pane 67
 - The Command Menu 59
- Converter 13
- Copy-Collection 89
- Copy-Object 21
- Copy-Self 21
- Count Down Space in Metering Partition 61
- Current-Number-Of-Elements 27, 89
- Current-Time 109
- Data Input 7, 38, 73, 75, 76, 81
- Debug-Format 113
- Debug-Heading 113
- Debug-Princ 113
- Debug-Print 113
- Debug-Terpri 113
- Declare_Operator 11
- Declare_Production 13
- Default 24, 33
- Default AutoSave Interval 63
- DefConstant 16
- Definition 38
- Definitions 41
- DefParameter 16
- Defsystem 97
- DeFuturing 1
- Delete 51
- Determined elements 27, 88, 89
- Discard 47, 66
- Display All Data Structures 60
- Do 17
- Dont-Optimise 109
- Downarrow 12
- Dribble File 59, 62
- Dribble Off 59
- Dribble On 59, 62, 63
- Dynamic Binding 13, 14, 15
- Eager 42
- Element-of 89, 90, 91
- Empty 23, 28, 33, 36, 44, 50, 88, 120
- Enable Breakpoints Within Abortable Code 60
- End 8
- EndIf 8
- EndLet 8

- Equal 21
- Equality Testing Operators
 - = 20, 117
 - =% 20
 - Not Equal sign 20, 117
- Equality-Tester 21
- Eval 1
- Evaluation 1
- Event 44, 46, 47, 58, 67
- Execute 17, 37, 43, 45, 46
- Expect 47
- Expectation Specifications
 - Active 51
 - Definitions 52
 - Delete 51
 - Timeout 52
- Expectations 47, 50
- Field Selection Operators 26, 36, 109
 - Centre Dot 25, 28, 29, 36
 - Circle Plus 28, 36
 - Is-Empty 36
 - Is-Not-Empty-or-Undefined 37
 - Is-Not-Undefined 36
 - Is-Undefined 36
- Field Testing Predicates
 - Is-Empty 36
 - Is-Not-Empty-or-Undefined 37
 - Is-Not-Undefined 36
 - Is-Undefined 36
- FIFO 64
- File Fonts 62
- Filter Statistics 63
- First-Of-Multiple-Values 111
- Flavors, system defined
 - Basic-Slot 37, 50
- Floated-Current-Time 109
- Focus-Node 5
- Fold 89
- Fold-Right 89
- Force 42, 43, 45, 48, 78, 79, 82, 117, 118
- Functional-Value 87
- Functions, system defined
 - All-Elements 88
 - All-Values-Of 111
 - An-Element 88
 - Apply-Function 90, 91, 109
 - As-Type 89
 - Associate 88
 - Bag 88
 - Breakpoint 109
 - Call-Function 109
 - Cardinality-Exceeds 89, 118
 - Coerce-To-Object 35, 109
 - Collection-Difference 89
 - Collection-Intersection 89
 - Collection-Is-Empty 89, 118
 - Collection-Is-Not-Empty 89, 118
 - Collection-Union 89
 - Comment 109
 - Copy-Collection 89
 - Current-Number-Of-Elements 27, 89
 - Current-Time 109
 - Debug-Format 113
 - Debug-Heading 113
 - Debug-Princ 113
 - Debug-Print 113
 - Debug-Terpri 113
 - Dont-Optimise 109
 - Element-of 89, 90, 91
 - First-Of-Multiple-Values 111
 - Floated-Current-Time 109
 - Fold 89
 - Fold-Right 89
 - Get-Property 89
 - Halt-Poligon 109
 - Has-Type 109, 117
 - Head 89, 90
 - Is-a-Bag 88
 - Is-A-Collection 90
 - Is-a-lazy-list 88
 - Is-a-lazy-list-tail 88
 - Is-A-Link-Cell 123
 - Is-A-Multiple-Values 111
 - Is-a-Set 88
 - Is-An-Ordered-Collection 90
 - Is-An-UnOrdered-Collection 90
 - Is-In 90, 117
 - Is-Not-In 90
 - Is-Not-Nil 110
 - Is-Of-Type 87
 - Is-Strictly-Of-Type 87
 - Join 90
 - Link-Between 123
 - Link-Between-Named 123
 - Link-Cell-Link 123
 - Link-Cell-Link-Name 123
 - Link-Cell-Linked-to 123
 - Make-a-Collection-of-Type 90
 - Make-into-lazy-list 88
 - Map-Over-A-Collection 90, 117
 - Multiple-Values 23, 111
 - Multiple-Values-Element-Of 111
 - Non-Strict-Type-Of 87
 - Number-of-Elements 90, 118
 - Optimise 110
 - Poligon-Format 46, 113
 - Poligon-Format-and-wait 113
 - Poligon-Princ 113

- Poligon-Princ-and-wait 113
- Poligon-Print 113
- Poligon-Print-and-wait 113
- Poligon-Terpri 113
- Poligon-Terpri-and-wait 113
- Reset-Poligon 58
- Rest 88
- Returns-Coerced-Result 110, 117
- Run-Poligon 57, 58, 110
- Second-Of-Multiple-Values 111
- Set 88
- Sort-Collection 90
- Stop-Poligon 110
- Strict-Type-Of 18, 87
- Tail 90
- The-Eighth 90
- The-Excluded-Subset 90
- The-Fifth 90
- The-First 90
- The-Fourth 90
- The-Last 90
- The-Ninth 90
- The-Second 90
- The-Seventh 91
- The-Sixth 91
- The-Subset 91
- The-Tenth 91
- The-Third 29, 91
- Trace-Format 113
- Unhalt-Poligon 110
- Uniquise 91
- With 91
- With-Printing-Bindings 110
- Without 91
- Without-Poligon-Clock 110
- Future 1, 18, 19, 20, 22, 26, 63, 66, 87, 110
- Future Force Rate 63
- Futures 115
- Garbage Collection 62
- General-Compile 97
- General-Compile-Load 97
- General-Compile-Load-Init 97
- General-Parse 97
- General-Parse-Init 97
- Get 89
- Get-Input-File-Name 40, 62
- Get-Next-Value 88
- Get-Property 89
- Ghost Elements 27
- Grammar 8, 13, 16, 101
- Graphics 68
- Halt-Poligon 109
- Hardcopy Minutes 65
- Has-Type 109, 117
- Head 36, 88, 89, 90
- identifier 101
- Identifiers 8, 9, 15
- Identifiers, special 23
 - Poligon-Blackboard 23, 33, 34, 35, 58, 113
 - The- 44
 - THE-class-name 44
 - The-Created-Node 24
 - The-Expected-Field 51, 52
 - The-Expected-Node 51, 52
 - The-Field 44
 - THE-field-name 44
 - The-Node 35, 44
 - The-Slot 44
 - The-Trigging-Node 44
- If 8
- Immediate 64, 66
- In Parallel For Each 50
- Include-Comment 109
- IndexedBy 28, 29, 33
- Inherit 120
- Initialisation 23, 24
- Input File Default 62
- Input Handler 35, 38
- Input Handler Class 35
- Input-Handler 35, 39
- Input-Procedure 38, 39, 76, 81
- InsertIf 27
- Inspect Details Of Structures 60
- Instance Creation 23, 42
- Instance-Of 30
- Instances 23, 34
- Internal Debug Messages Enabled 62
- Is-a-Bag 88
- Is-A-Collection 90
- Is-a-lazy-list 88
- Is-a-lazy-list-tail 88
- Is-A-Link 120
- Is-A-Link-Cell 123
- Is-A-Multiple-Values 111
- Is-a-Set 88
- Is-An-Ordered-Collection 90
- Is-An-UnOrdered-Collection 90
- Is-Empty 36
- Is-In 90, 109, 117
- Is-Not-Empty-or-Undefined 37
- Is-Not-In 90
- Is-Not-Nil 110
- Is-Not-Undefined 36
- Is-Of-Type 87
- Is-Strictly-Of-Type 87
- Is-Undefined 36

- Is_a_Prefix 17
- Join 90
- Jump the clock when idle 61
- ket 101
- KeyedBy 28, 33
- Keyword 2, 7
- Knowledge Base 4, 7, 58
- Knowledge Source 4, 7, 41, 43, 44, 45, 77
- L100 37
- L100 Mode 93, 94
- Lambda 71
- Lambda Binding 13, 14
- Latest 2
- Lazy 15
- Lazy Evaluation 41, 42
- Lazy Function Arguments 42
- Lazy List 18, 22, 42, 85, 87, 88, 90
- Lazy list processing functions
 - Is-a-lazy-list 88
 - Is-a-lazy-list-tail 88
 - Make-into-lazy-list 88
 - Rest 88
- Lazy-List 90
- Left 66, 67
- Let 8, 14, 71, 109, 110
- Levels 29, 30
- Lexical Definition 14
- LIFO 64
- Link 120, 121
- Link Subsystem 120, 121, 122
- Link-Between 123
- Link-Between-Named 123
- Link-Cell 87, 120, 123
- Link-Cell-Link 123
- Link-Cell-Link-Name 123
- Link-Cell-Linked-to 123
- Link-Cells, functions for processing
 - Is-A-Link-Cell 123
 - Link-Cell-Link 123
 - Link-Cell-Link-Name 123
 - Link-Cell-Linked-to 123
- Links, functions for processing
 - Link-Between 123
 - Link-Between-Named 123
- Lisp pane 59, 113
- List 88, 90
- Lists 8, 12, 13, 71, 72, 85
- Literal 17
- LL1 8
- Load-Into-ZMacs 97
- Log 59, 62
- Log File 59, 62
- Log Messages Lasting Longer than N ms 65
- Loop 110
- Lozenge 8
- lsq 101
- Make-a-Collection-of-Type 90
- Make-into-lazy-list 88
- Make-System 7, 57
- Make_Prefix 17
- Map-Over-A-Collection 90, 117
- Member 22
- Message Pane 59, 63, 66
- Message Time Log File 65
- Meta-Left 66
- Meta-Middle 66, 68
- META-name 23, 31, 32
- Meta-Right 67
- Meta-X Compile Buffer 57
- Meta-X Compile Region 57
- Meta-X Compile Region/Buffer 57
- Metaclass 31
- Meter Micro Enables 61
- Metering 61
- Middle 66, 68
- MIMD 4
- Mode Line 69, 73, 95
- Model 2
- Modify 2
- ModifyWith 28
- Mouse Button Selections
 - Left 66, 67
 - Meta-Left 66
 - Meta-Middle 66, 68
 - Meta-Right 67
 - Middle 66, 68
 - Right 67, 68
- Mouse-1-R Compile Region 94
- Multiple-Values 22
- Multi-Future 87
- Multiple Values 23, 38, 41, 111
- Multiple-Values 20, 23, 29, 87, 110, 111
- Multiple-Values-Element-Of 111
- MultipleLet 23, 109, 110
- Name 34
- New Instance Of 23, 44, 54
- No 21
- Node 2
- Nodes 34
- NoEvent 39, 48
- Non-Nil-Collection 87
- Non-Strict functions 19
- Non-Strict-Type-Of 87
- Not_Equal 10
- Nth 89

- number 101
- Number-Of-Elements 27, 90, 118
- Number-of-instances 34
- Number-of-subsystems 34
- Number-of-supersystems 34
- Numbers 8
- Operators 7, 8, 9, 10, 11, 13, 17, (See Equality Testing Operators), (See Field Selection Operators), (See Field Selection Operators), (See Field Testing Predicates), (See Update Operators), (See Update Operators), (See Update Operators), 71, 72, (See Operators, system defined), (See Operators, system defined), (See Operators, system defined), (See Equality Testing Operators), (See Operators, system defined), (See Update Operators)
- Operators, system defined
 - As-Type 89
 - Cardinality-Exceeds 89
 - Collection-Difference 89
 - Collection-Intersection 89
 - Collection-Union 89
 - Is-In 90, 117
 - Is-Not-In 90
 - Is-Of-Type 87
 - Is-Strictly-Of-Type 87
 - With 91
 - Without 91
- Optimise 110
- Ordered-Collection 87, 89, 90
- Otherwise Part 44, 78, 79
- Output routines
 - Debug-Format 43, 113
 - Debug-Heading 113
 - Debug-Princ 113
 - Debug-Print 113
 - Debug-Terpri 113
 - Poligon-Format 46, 113
 - Poligon-Format-and-wait 113
 - Poligon-Princ 113
 - Poligon-Princ-and-wait 113
 - Poligon-Print 113
 - Poligon-Print-and-wait 113
 - Poligon-Terpri 113
 - Poligon-Terpri-and-wait 113
 - Trace-Format 113
- Parallel 60
- Parallel Clock Rate 63
- Parallel Metering Slow-Down Factor 61
- ParallelLet 115
- Parameter Lists 8, 15, 71
- Parameters 59, 60, 65
- Parse 85
- Parse and Print Buffer 93
- Parse and Print Region 93, 94
- Parse Buffer 93, 94
- Parse File 94
- Parse into Buffer 94
- Parse Region 93, 94
- Parse_a_list 85
- Parse_in_Context 85
- Pascal 27, 30
- Percent 12
- Poligon 50
- Poligon Package 18
- Poligon-Blackboard 23
- Poligon-Format 113
- Poligon-Format-and-wait 113
- Poligon-Princ 113
- Poligon-Princ-and-wait 113
- Poligon-Print 113
- Poligon-Print-and-wait 113
- Poligon-Terpri 113
- Poligon-Terpri-and-wait 113
- Poligon-User Package 18
- Pragmata 15
- Process 4
- Processors 3
- Property Inheritance 67, 120, 121
- Property Inheritance through Links 120
- Property Inheritance through Supersystems 119
- Property Inheritance, types
 - Compile-time Property Inheritance 119
 - Property Inheritance through Links 120
 - Property Inheritance through Supersystems 119
 - Run-time Property Inheritance 119
- Punting 64, 118
- Quoted-Form 87
- Random 64
- Record Message Times In List 65
- Record Message Times In Log File 65
- Recycle 47
- Redisplay 65
- Redisplay Class Pane On Creation 63
- Remote Address 18, 28
- RemoveIf 27
- Reset-Poligon 58
- Rest 88
- Returns-Coerced-Result 110, 117
- Right 67, 68
- Root 23, 33, 58, 75, 81
- rsq 101
- Rule 41, 43

- Rule Header 43, 44, 45, 52, 78
- Run 58, 65, 110
- Run-Polygon 57, 58, 110
- Run-time Property Inheritance 119
- Run-Time System 3, 7, 18, 57
- Scheduler 63, 64, 66
- Scheduler options
 - FIFO 64
 - Immediate 64, 66
 - LIFO 64
 - Random 64
- Second-Of-Multiple-Values 111
- Select Part 45, 46, 67
- Selected-Circuit 110
- Serial 65
- Serial Clock Rate 62
- Set 18, 25, 87, 88
- Set processing functions
 - Is-a-Set 88
 - Set 88
- Setf 11
- Simulation Parameters
 - Abort After Input File Closed 63
 - Action on Deadlock 61
 - Allow All Within Abortable Code 60
 - Automatic GCing 62
 - Break Clock Ticks 63
 - Break Message Punting 64
 - Break Signal Records 63
 - CARE Debug History 62
 - Compilation Factor 61
 - Count Down Space in Metering Partition 61
 - Default AutoSave Interval 63
 - Display All Data Structures 60
 - Dribble File 59, 62
 - Enable Breakpoints Within Abortable Code 60
 - File Fonts 62
 - Filter Statistics 63
 - Future Force Rate 63
 - Hardcopy Minutes 65
 - Input File Default 62
 - Inspect Details Of Structures 60
 - Internal Debug Messages Enabled 62
 - Jump the clock when idle 61
 - Log File 59, 62
 - Log Messages Lasting Longer than N ms 65
 - Message Time Log File 65
 - Meter Micro Enables 61
 - Metering 61
 - Parallel Clock Rate 63
 - Parallel Metering Slow-Down Factor 61
 - Record Message Times In List 65
 - Record Message Times In Log File 65
 - Redisplay Class Pane On Creation 63
 - Scheduling Strategy 64
 - Serial Clock Rate 62
 - Start Metering After Data Time 61
 - Statistics File 62, 65
 - Stop At Or After Signal Record 64
 - Stop At Or After Time 64
 - Switch Dribble On When You Run 63
 - Trace Clock Ticks 63
 - Trace If Segment Takes Longer Than n usecs 60
 - Trace Message Punting 64
 - Trace Messages 63
 - Trace Node Creation 64
 - Trace Rules 63
 - Trace Signal Records 63
 - Trap On Errors Within Abortable Code 60
 - Which Instrument To Use 64
 - Window Fonts 62
- Slot 2, 30
- Sort 90
- Sort-Collection 90
- SortedBy 28, 33
- Special 2
- Start Metering After Data Time 61
- Statistics 62, 65
- Statistics File 62, 65
- Stop At Or After Signal Record 64
- Stop At Or After Time 64
- Stop-Polygon 110
- Strict functions 19
- Strict-Type-Of 18, 87, 88
- String-Append 9
- Strings 8, 9, 13, 17, 71, 72
- Structure 16, 17, 19, 20, 73, 80
- Subclass 30, 31, 74
- Subset 25
- Subst 22
- Subsystem Of 24
- Subsystems 31, 34
- Superclass 30
- Supersystems 31, 34
- Switch Dribble On When You Run 63
- System Defined Data Structure Types
 - Bag 18, 25, 26, 27, 77, 78, 79, 87, 88, 90
 - Future 1, 18, 19, 20, 22, 26
 - Lazy List 18, 22, 42, 87, 88, 90
 - Link-Cell 120, 123
 - Remote Address 18
 - Set 18, 25, 87, 88

- System Defined Types
 - Collection 87
 - Functional-Value 87
 - Future 87
 - Link-Cell 87
 - Multi-Future 87
 - Multiple-Values 29, 87
 - Non-Nil-Collection 87
 - Ordered-Collection 87, 89, 90
 - Quoted-Form 87
 - Type-Specifier 87
- System Field Names 30
 - Clock 34
 - Inherit 120
 - Instance-Of 34
 - Instances 34
 - Is-A-Link 120
 - Links 120
 - Name 34
 - Number-of-instances 34
 - Number-of-subsystems 34
 - Number-of-supersystems 34
 - Subsystems 34
 - Supersystems 34
- Tab Key 93
- Tags Compile Changed Definitions 94
- Tail 18, 88, 89, 90
- Tail Recursion Optimization 110
- The 117
- The Command Menu 59, 65
- The Rule Monitor Pane 67
- The- 44
- THE-class-name 44
- The-Created-Node 24, 44
- The-Eighth 90
- The-Excluded-Subset 90
- The-Expected-Field 51, 52
- The-Expected-Node 51, 52
- The-Field 44, 47
- THE-field-name 44
- The-Fifth 90
- The-First 90
- The-Fourth 90
- The-Last 90
- The-Ninth 90
- The-Node 35, 44
- The-Second 90
- The-Seventh 91
- The-Sixth 91
- The-Slot 44
- The-Subset 91
- The-Tenth 91
- The-Third 29, 91
- The-Triggering-Node 44
- Time-Of-Input-Record 38
- Timeout 52
- Timeout Part 42, 52, 68
- Toplevelstatement 16
- Topology 3
- Total-Word-Size 21
- Trace & Break 118
- Trace Clock Ticks 63
- Trace If Segment Takes Longer Than n
usecs 60
- Trace Message Punting 64
- Trace Messages 63
- Trace Node Creation 64
- Trace Rules 63
- Trace Signal Records 63
- Trace-Format 113
- Trap On Errors Within Abortable Code 60
- Type Checking 15
- Type manipulating Functions and Operators
 - Is-Of-Type 87
 - Is-Strictly-Of-Type 87
 - Non-Strict-Type-Of 87
 - Strict-Type-Of 87
- Type-Specifier 87
- Types 87
- Undetermined elements 27
- Unhalt-Polygon 110
- Uniquise 91
- Unless 24, 25, 44
- Unlink 122
- Unlink Subsystem 121, 122
- Uparrow 12
- Update 43, 47, 48, 50, 78
- Update Operators 43, 46, 48, 49, 53, 54,
117
 - " 78, 82
- Update types
 - Modify 2
- Updated Class Fields 24, 25, 44
- User Defined Function Names
 - Get-Input-File-Name 40, 62
 - Input-Procedure 38, 39, 76, 81
 - Time-Of-Input-Record 38, 76, 81
- User Defined Initialisation 7, 35, 39, 75,
81
- User-Abort-Function 40
- Value List 2, 23, 27
- When Part 45
- Which Fails 26
- Which Instrument To Use 64
- Which Satisfies 26
- Window Fonts 62
- With 91
- With-Printing-Bindings 110

Without 91
Without-Polygon-Clock 110
Zmacs 57, 95
→ 11
↓ 12
• 11, 25, 28, 29, 36, 71
“+” 25
≠ 12, 22
≠ 20, 117
⊕ 28, 36
√ 101
◇ 8

Problems with Problem-Solving in Parallel: The Poligon System

by
James Rice
(Rice@Sumex-Aim.Stanford.Edu)

**Knowledge Systems Laboratory
Stanford University
701 Welch Road
Palo Alto, CA 94304**

The author gratefully acknowledges the support of the following funding agencies for this project; DARPA/RADC, under contract F30602-85-C-0012; NASA, under contract number NCC 2-220; Boeing Computer Services, under contract number W-266875.

Abstract

This paper describes the desire to speed up programs in the field of Artificial Intelligence through the use of parallel hardware architectures and why this objective is not a simple one to achieve.

Poligon, a system designed to investigate ways to mount Artificial Intelligence programs on parallel hardware, is described, experiments performed to date on this system are described and tentative results are given.

Achieving useful speed-up has proven very difficult. These difficulties are enumerated and explained.¹

1. Introduction

The domain of supercomputing has traditionally been very large regular problems. This has been driven by two main forces;

- A large class of important problems were soluble by existing programming technology but were intractable with "normal" processors, e.g. PDE solution, finite element analysis or simulation.
- Early programming languages focused on Arrays as data structures, whose use could efficiently use the hardware available. This led to the idea of vector and array processors.

It is, therefore, by no means a coincidence that the sort of problems that tend to use existing supercomputers are those problems best suited to supercomputers.

The field is changing now, however. This is driven by two main forces;

- Developments in hardware technology now allow the development of multiprocessor systems composed of large numbers of relatively simple processors, which are potentially more cost effective than existing super-complex supercomputer uniprocessors.
- Both hardware and software technologies have progressed to a point where a number of problems which have become soluble by means of symbolic programming would now like a slice of the speed-up cake.

Symbolic computation has for a long time been accused of inefficiency. Recent developments in compiler and hardware technologies, however, have allowed the development of high performance uniprocessor workstations for the execution of symbolic programs. These have shown that there is a

¹This paper also appears in the proceedings of the *Third International Conference on Supercomputing, May 1988, and Artificial Intelligence, Simulation and Modelling*, Lawrence Widman (ed), John Wiley Publishing Company, New York 1989

large class of *Artificial Intelligence (AI)* problems for which significantly greater computational resources will be needed to make these problems worth addressing. This has focused the attention of AI and symbolic programming research on the exploitation of parallelism.

The sort of problem currently applied to supercomputers is very crystalline [Seitz 85] in nature. This means that a relatively small "inner loop" of the computation can be vectorized in order to exploit existing supercomputer hardware [Kuck 81]. Similarly such problems can often exploit parallelism at a finer grain in a systolic manner [Kung 78].

AI problems have none of these useful characteristics [Lee 85]. This paper describes first what is meant by *Problem-Solving* and how this relates to parallelism (Section 2). It goes on to describe

Poligon [Rice 86] a system implemented in order to investigate the potential for speed-up of a class of AI applications called "Blackboard Systems" through parallelism (Section 3). After this some preliminary experiments and what we have learned from them and discussed (Section 4).

2. Parallelism and Problem-Solving

In this section we examine what is meant by "Problem-Solving", contrasting it with common supercomputing doctrine and concerns. This will show why it is that a different approach to parallelism than is taken by conventional programs is necessary in *AI* and also why it is so hard to achieve.

2.1. What is "Problem-Solving"?

Questions are never indiscreet. Answers sometimes are.

— Oscar Wilde, *"An Ideal Husband"*

"Problem-Solving" was often taken to refer to the process of searching a tree or graph of alternative solutions. "Knowledge" is that which allows the program to eliminate searching parts of the tree. For instance, a chess playing program might have a tree made of all of the legal moves at any given point¹. The term "knowledge" will always be used in this sense in this paper. The application of strategic knowledge, such as knowledge about chess end games, to each generated node in the tree would point out to the system likely candidate paths to follow. The method of constructing all legal possibilities at any given leaf of a dynamically generated tree and then testing them to determine whether they are possibilities worth following is usually referred to as the *Generate and Test* method. It is an axiom of such systems that the more knowledge there is the less blind search has to be done - the more efficiently the tree is pruned.

The focus of much AI research is on the use of knowledge to reduce or obviate search. This is because such searches are expensive and combinatorial processes. The use of knowledge in this way might not be the best solu-

¹Clearly this tree cannot be fully instantiated with the resources available in the universe.

tion for the future since the use of highly parallel architectures to evaluate multiple alternatives might be faster than executing this highly specialized knowledge. What is more, this could also save the human cost of acquiring and encoding such knowledge. The acquisition of knowledge is generally thought to be one of the major obstacles in the way of the more general application of AI systems to real-world problems.

The important thing, for the purpose of this paper, about problem-solving systems and the problems that they address is that they are structurally different from "*conventional*" programs. Throughout this paper the terms "Problem-Solving" and "AI system" will be used to describe these systems. The term "Conventional" will be used to describe existing practice in the supercomputer world. Some of the characteristics that make such a problem different from a conventional programming problem are listed below.

- The problem itself is often ill-defined.
- There is often more than one possible solution. This means that a satisficing¹, rather than an optimal solution is usually the "right" answer. This is quite unlike most conventional programs for which there is one and only one right answer, within the margin of error of the system.²
- The paths to a solution cannot predefined in such systems. Possible solution paths must be dynamically generated and tried.
- The structure of such programs differs from conventional programs in three fundamental ways; in their data structures, their control flow and their control structures.

Data Structure It is generally the case that the data upon which the system has to operate cannot be encoded simply into an array. This is because such data structures are usually highly complex and often cyclic graphs, which are created dynamically, thus precluding static allocation and optimization.

Control Flow The solution to the problem is not regular, which is to say that the behavior of the problem-solver is typically very data-dependent. In a PDE solving program, for instance, the computational demands of the system at any point are well understood. This is because well defined and well understood algorithms are used and the computational demands of matrix inversion, for example, are reasonably easy to estimate. This is not the case in AI programs. Apparently trivial changes to the source data can cause huge changes to the

¹A solution that is said to be "*good enough*."

²Linear optimization is a notable exception to this. Clearly many programs use heuristics and so the distinction made here is simply one of degree. AI problems are usually composed of a larger proportion of heuristics than conventional programs.

computation performed. As an example of this one might consider the behavior of a chess program when the opponent elects to make an unexpected move. What is more, the code generated for these programs is usually very branchy [Lee 85], thus reducing the benefits of fine grained pipe-lining.

Control Structures The knowledge that AI programmers try to encode in their programs is usually functionally different from that knowledge which is usually encoded in conventional programs. That is to say it is more likely to be a high-level specification of the intended behavior of the system, as opposed to a set of instructions for how to compute the answer. Such details are usually left to the system. For instance, the program might be compiled into a set of assertions and rules in a Prolog system [Clocksin 81]. The program itself is executed indirectly through a virtual machine which interprets these specifications as its instructions. This results in most of such programs not being amenable either to existing vectorizing algorithms or to the application of well defined algorithms.¹

The factors mentioned above result in AI problems not having the properties needed for them to be parallelized by conventional means. This is cause for considerable concern for those who would like to achieve orders of magnitude of speed-up for their AI programs.

2.2. Concerns for Supercomputers

On how to trap a lion in a desert [Petard 38]: A topological method.

We observe that a lion has at least the connectivity of the torus. We transport the desert into four-space. It is then possible [Seifert 34] to carry out such a deformation that the lion can be returned to three-space in a knotted condition. He is then helpless.

Implementors and programmers of supercomputers have traditionally focused on the efficient use of the hardware and the matching of the hardware to the problem. Some examples of these are discussed below.

2.2.1. Where does parallelism come from?

Parallelism in conventional programs is either easy to get or nearly impossible. If the program does a lot of simple operations on arrays whose dependencies and recurrences are simple and can be unraveled then massive data parallelism² can be exploited. It is by this means that vector machines are able to achieve their performance. It is not generally the case that there is, qualitatively speaking, more than one thing happening at any given

¹These interpreters themselves may, however, be implemented using well understood algorithms or microcode.

²Parallelism due to similar operations being performable on independent items of data, for instance elementwise addition of two arrays.

time. Such programs are parallel in a SIMD sense [Flynn 72]. If the control flow is too complex to analyze then the compiler may not be able to unwind the parallelism out of the program.¹

AI programs are typically short on data parallelism. There are certainly problems which have significant data parallelism but not of the order that one might get in extremely regular, conventional programs. This means that an AI system which hopes for speed-up through parallelism must be able to exploit *knowledge parallelism*. It must be able to execute a significant number of different chunks of the program simultaneously. This is MIMD parallelism. The Poligon system described in Section 3 is designed to exploit this sort of parallelism.²

Most high performance processors today exploit pipe-line parallelism in the execution of instructions. Pipe-line parallelism is also exploited at a somewhat coarser grain by the new generations of multiprocessor systems such as systolic arrays. It is crucial that any system hoping to exploit parallel hardware effectively should be able to exploit pipe-line parallelism. This is, in fact, considerably harder in AI systems because of the irregular structure of the problem. The Poligon system tries wherever it can to exploit pipe-line parallelism.

2.2.2. What sort of hardware should be used?

In order to be able to exploit the parallelism in a program to the best possible degree there must be an appropriate match between the compiled program and the target hardware. This means that if a speed-up of no more than 10 to 20 is either hoped for or expected then the program should probably be executed on a shared-memory multiprocessor³. If more speed-up than this is needed then a hardware design that will scale better should be used - some form of distributed memory architecture.⁴ This could, in practice, have a grain size varying from that of the *Cosmic Cube* [Seitz 85] to that of the *Connection Machine*TM [Hillis 85].⁵ The Poligon system is designed to be matched to run on a multiprocessor, which should scale satisfactorily to the order of hundreds or thousands of processing elements, each element being a highly competent symbolic language processor. This is the pure

¹The Connection Machine [Hillis 85] is an example of an experiment to test the contrary hypothesis, that SIMD machines are, indeed, appropriate for AI applications.

²MIMD programs typically have a set of implementation difficulties and bugs which are not so frequently seen in SIMD programs. These are caused by having a number of radically different types of program executing, all at different speeds and trying to communicate with one another. This causes data to arrive "out of order" and race conditions. Many of the pit-falls of parallel AI programming mentioned in this paper are a consequence of this.

³Some experiments have shown rather disappointing results here, saying that this is all that can really be hoped for. [Gupta 86]

⁴Recent claims have been made that some shared memory architectures can scale well [Wilson 87].

⁵Connection Machine is a trade mark of the Thinking Machines Corporation.

value passing *CARE* machine model [Byrd 87], one of several *CARE* machine models implemented as part of the same project of which Poligon is a part.

2.2.3. Compilation

Vectorizing FORTRAN compilers have been the main implementation language in supercomputing circles for quite some time. There is considerable inertia in the field in this respect. Similarly AI programmers are in many senses locked into the use of Lisp [Steele 84] or Prolog [Clocksin 81] as their implementation languages. Problem-Solving systems have traditionally not been very efficiently implemented, even if the underlying implementation language has been. This is because it is expensive in human terms to implement such systems efficiently and their typical life span has not justified this sort of optimization effort. This state of affairs is beginning to change. There is now a demand for highly competent programs using AI techniques being embedded, for instance, into military hardware. This asks not only for high performance but also for high reliability, maintainability and modifiability. Lisp and Prolog in their common implementations are not languages which can easily be parallelized in the same way that FORTRAN compilers are.¹ There is, therefore, a need to develop languages not only capable of exploiting the parallelism in forthcoming hardware but also capable of expressing the richness of these complex symbolic programs. On top of these will need to be built highly competent tools and frameworks which will be needed for a satisfactory parallel AI development environment. The Poligon system is a first-cut prototype system developed with the objective of being able to extract parallelism from programs both by the system and by encouraging a clear programming style and problem decomposition methodology, which leads to more parallel programs.

2.3. Concerns for Problem-Solvers

The concerns of the implementors of Problem-Solving systems are quite different from those of supercomputer programmers. Some of these concerns are enumerated below.

2.3.1. Solution quality

As has been mentioned above, AI programs are generally expected to produce a satisficing solution. This has a significant impact on the behavior of the program, since paths used to determine heuristic solutions might be very different from those used to find analytic solutions, even if analytic solutions are known.

¹Implementations of both of these languages have been made with "do-this-bit-in-parallel" constructs e.g. [Gabriel 84] and [Clark 85] and much work is now focusing on the automatic extraction of parallelism in these languages but as yet no symbolic programming equivalent of a vectorizing FORTRAN compiler has been produced. This is because it is generally not known at compile time whether any given expression is worth evaluating in parallel, given the costs of process creation and such-like.

2.3.2. Search

These heuristic programs are typically characterized by searching a great deal for patterns over a large graph.¹ This large amount of search admits both *And* and *Or* parallelism, in principle. The Polygon system has specific mechanisms to facilitate the efficient execution of such searches.²

2.3.3. Coherence

The implementor of an AI program may not be aware of the eventual behavior of his program when he is implementing it. This is a function of the complex nature of such problems and the fact that the paths to their solutions are not predefined. It is, nevertheless, very important that the program reach a coherent solution, even if just a satisficing one. It is no good if different parts of the solution space have mutually contradictory local solutions which contribute to the overall solution. Because the knowledge that goes into such systems is usually implemented in distinct chunks, which may know little about the operations performed by other such chunks, there is significant potential for the system getting confused as different subsystems "trample on each others' toes." This means that it is by no means a trivial issue to make sure that a coherent or convergent solution is achieved by Problem-Solving systems. This problem is exacerbated by the asynchronous behavior which can happen in MIMD parallel systems. The Polygon system is designed to help the programmer arrive at coherent solutions, whilst still encouraging parallelism at a fine grain.

2.3.4. Programming

Heuristic programs are typically large and their density is great.³ This means that their code encapsulates a great deal of knowledge. It is difficult to write such programs for a number of reasons.

- It is difficult to acquire the knowledge that goes into them, since this is typically not encoded already in a formal algorithmic way.
- It is difficult to represent the knowledge once it has been acquired. For instance, the programming associated with implementing a statement such as "*Control of center is very important during openings*" would be considerable.
- Good, clean implementations of such systems need to maintain the logical independence of the knowledge in the system. This is because failure to do so can result in systems that are very brittle when

¹In fact this graph can be of semi-infinite size and often has to be computed on demand; cf. the game tree for a chess game.

²If search dominates the computation then massively parallel machines such as the Connection Machine [Hillis 85] may well prove to have the best performance.

³This means that the number of executed machine instructions for each line in the source code is typically very large.

knowledge is executed in new orders or when new knowledge is added. The interconnectedness of knowledge is often difficult to determine when the knowledge is formulated. Clearly having dependencies between pieces of knowledge could have a significant impact on the amount of parallelism that could be extracted from such a program and on the program's ability to get the "right" answer.

It is, therefore, a major concern of AI programmers that these programs should be easy to implement, debug, modify and maintain.

3. Poligon a System for Parallel Problem-Solving

In this section we describe Poligon. Poligon is an attempt to produce a system which addresses the issues mentioned above to support the development of parallel AI systems. It represents, in many ways, an attempt to find an analogue for and implement a parallel form of existing AI systems, known as *Blackboard Systems* [Nii 86].

A brief description of the important aspects of blackboard systems will be given, then Poligon itself will be described; structurally, in the way in which it matches its problem domain, and the way in which it is matched to its target hardware.

3.1. Blackboard Systems

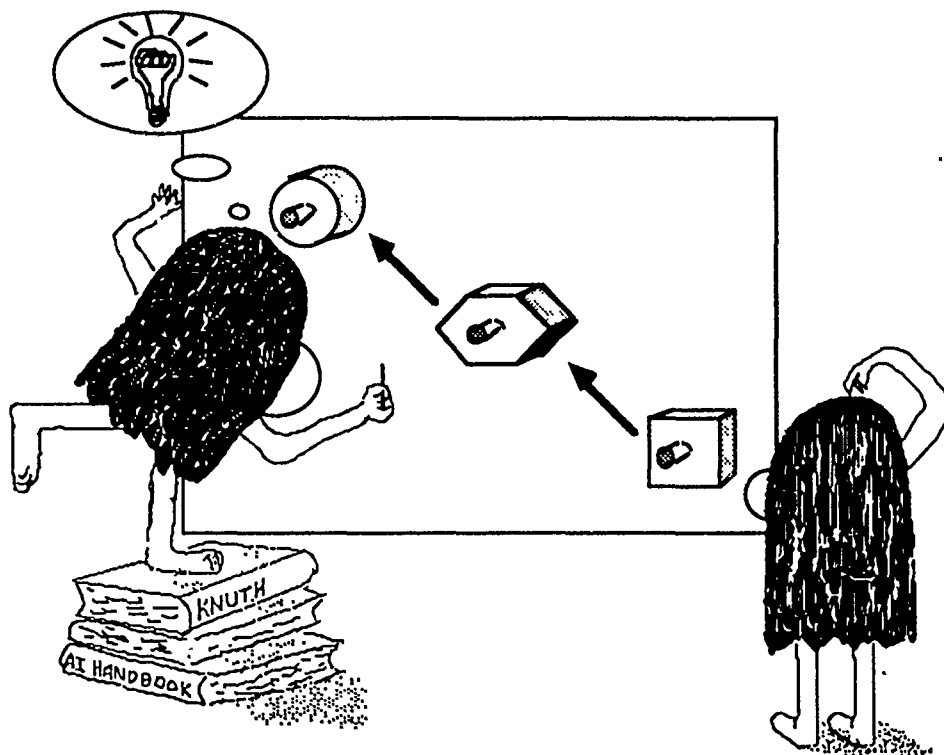


Fig. 1. The Blackboard Metaphor. *Eegar*, uses encoded knowledge and comes to a startling conclusion.

Blackboard systems are instances of a particular computational or problem-solving model — the "blackboard" model or metaphor. This metaphor takes as its source the idea of a collection of experts gathered around a blackboard (see Figure 1). Each expert has a specific domain of expertise, which relates to how a part of the problem at hand is to be solved. Each expert looks at the blackboard for representations of the problem which are of interest to his specific area of expertise. Having found such a piece of information he performs whatever operations he finds necessary and posts his conclusions on the blackboard. This new representation of part of the solution might itself be of interest to another expert and so the process continues.

It is clear from this that the sum of the knowledge in the system must be sufficient to connect all of these areas of expertise. With less knowledge than this the problem simply will not be soluble. With more knowledge than this it should be possible to achieve successively higher performance from the system; be it faster solutions or better solutions.

This simple model has considerable intellectual appeal and has been the cause of substantial research. It is often claimed that all of these "experts" should be able to operate simultaneously. The Polygon system represents an attempt to test this assertion.

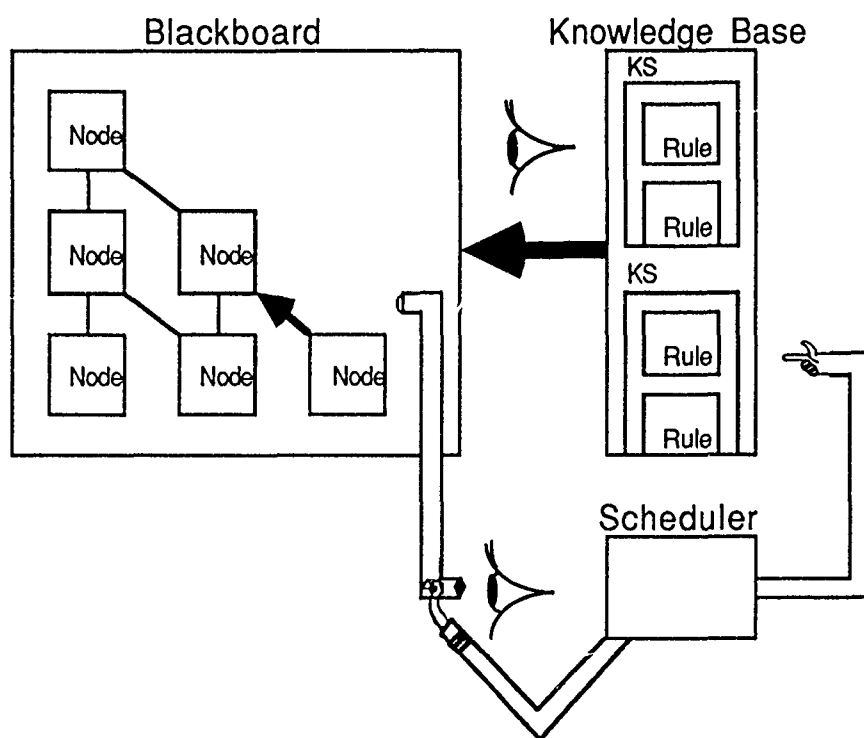


Fig. 2. A Serial Blackboard System. Here, the Scheduler notices a modification event and invokes a Knowledge Source.

Blackboard systems are typically implemented as large data structures — the blackboard — in which are stored the elements of the possible solutions, called *Nodes*, which are typically linked together in some way to form a

complex graph. There are normally a large number of these nodes, representing everything from the input data through intermediate solutions to high level abstractions of the current state of the solution. Nodes have internal structure, which allows the mapping of names onto values. They are usually made up of a collection of named *Slots* or *Fields*, which contain data pertinent to the solution. The knowledge in the system is usually implemented as a collection of pattern-action *Rules* collected into groups called *Knowledge Sources (KSs)* [Nii 80]. These reside in an area referred to as the *Knowledge Base* (see Figure 2).

3.1.1. Consistency and Coherence

Reaching a coherent solution, discussed in Section 2.3.3, in a blackboard system is a function of achieving consistency in a number of aspects:

Node Level The program should create the right number of nodes representing the elements in the solution and they should be connected together correctly.

Slot Level The slots in the nodes should contain a respectable representation of the state of that node and its relationship to others.

Rule Execution When rules are executed they should do so in an environment which is internally consistent. This means that any information used in the rule during its execution should be based on a consistent snapshot of reality.

3.2. A description of Poligon

Poligon is a framework for the development of blackboard-like applications on a (simulated) multiprocessor. It consists of:

- 1 A compiler, which compiles a high-level description of the Blackboard's structure and the knowledge to be applied by the system, to run on a distributed memory multiprocessor.
- 2 A run-time system which provides a debugging and testing environment for Poligon programs as well as run-time support.

Both the compiler and the run-time system are thoroughly integrated with the program development environment of ExplorerTM Lisp machines¹, the machine on which the execution of Poligon programs are simulated.

Serial blackboard systems are implemented with the nodes being represented as records on the blackboard.² The knowledge is encoded in knowledge sources. These are typically compiled into procedures which are in-

¹Explorer is a trade mark of Texas Instruments Incorporated.

²These records might well be Pascal-like records or instances of some class in the native system's object-oriented package.

voked by the blackboard system's kernel. There is some form of scheduler for the knowledge, which invokes one knowledge source after another. The blackboard and the knowledge base both share the same address space, though they are functionally distinct. Knowledge sources are "invoked" (executed) as a result of changes in the blackboard placing that change event in a queue used by the scheduler. The scheduler repeatedly picks a knowledge source which is interested in the type of event at the end of the queue.

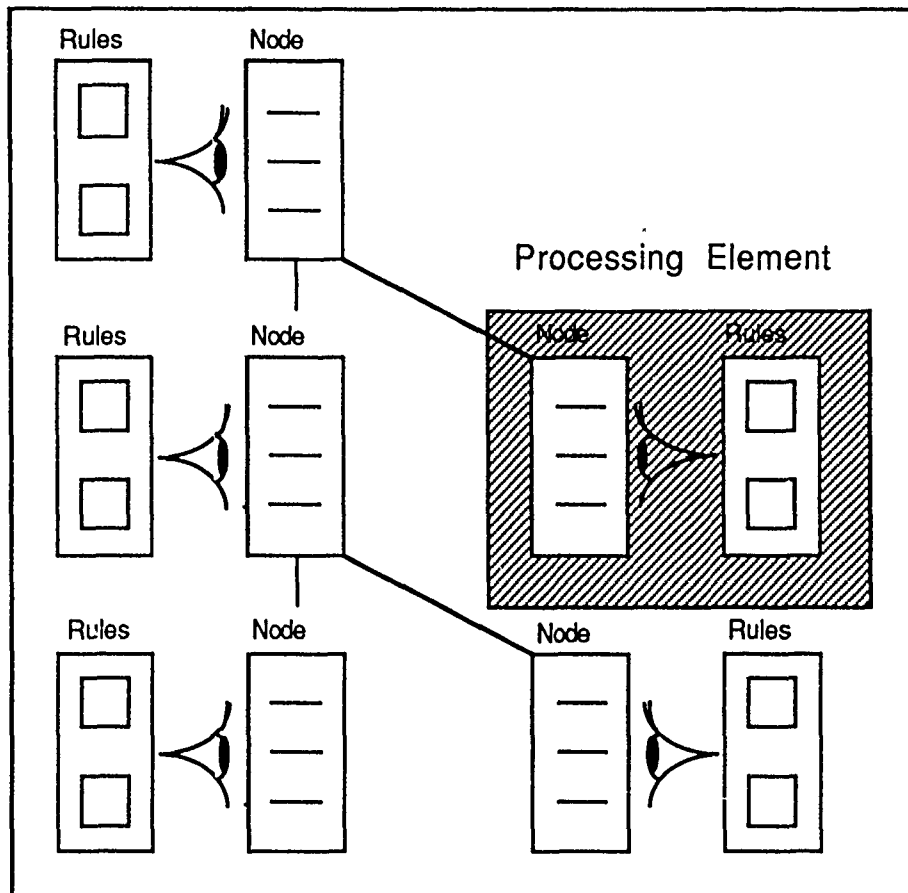


Fig. 3. Polygon's Blackboard. Nodes are seen linked together being watched by Rules, waiting for modification events.

The design of Polygon has been motivated by the idea of trying to eliminate the bottlenecks that would be experienced if an existing, serial blackboard system were to be parallelized by the inclusion of do-this-bit-in-parallel constructs.¹ The major changes from this model are listed below.

- The scheduling queue of a serial system is eliminated altogether in Polygon. This means that concurrent attempts to invoke rules are not held up waiting for access to this shared data structure.

¹The Cage system [Aiello 86] is an example of a considerably more conservative approach to the parallelizing of blackboard systems.

- Having a knowledge base, which is logically distinct from the blackboard, is no longer necessary since there is now nothing to get between them to control the application of the knowledge. This allows all knowledge to be attached to those nodes that are interested in the knowledge by the compiler (see Figure 3).

These changes eliminate at one stroke the bottlenecks of the shared scheduler and the knowledge base to blackboard interface. These changes allowed the development of the idea of the "node-as-a-processor" metaphor for parallel blackboard systems.

Having eliminated the scheduling mechanism, however, one needs some means of determining when a certain piece of knowledge should be invoked. It would be hopelessly inefficient to have all of the knowledge executed all of the time, since most of the time it would find itself inapplicable. It was decided that a simple *daemon-driven* approach would be used to avoid this problem. This results in the knowledge being directly sensitive to changes in the blackboard and able to act immediately upon any such changes.

Existing blackboard systems often express the knowledge in their knowledge sources as collections of pattern-action rules. These are normally executed serially, in the lexical order in which they are defined. Polygon on the other hand compiles knowledge sources away all together, allowing their constituent rules to be executed in parallel.

The node-as-a-processor metaphor is itself a major step away from the normal means of implementing blackboard systems. This, however, is not enough. This would give us data parallelism, resulting from the large number of nodes in the system being able simultaneously to execute rules, whilst still failing to exploit the potential knowledge parallelism. This is because each processing element is a uniprocessor, clearly capable of executing at most one rule at a time.¹ Polygon, therefore, goes beyond this simple model to one which would more accurately be called the "rule-invocation-as-a-process" model. This allows the Polygon system to distribute concurrent rule invocations to different processing elements (see Figure 4).

The elimination of serializing components in a blackboard system also eliminates those mechanisms which are normally used to preserve coherency in the solution. Clearly there is a trade-off which can be made between the amount of control and coherency preserving mechanisms and the amount of exploitable parallelism. Polygon is an experiment to explore one extreme of this spectrum. It remains to be seen whether the trade-off made in Polygon results in an overall improvement in system performance.

¹Each element allows multiple processes but only one is executed at any time.

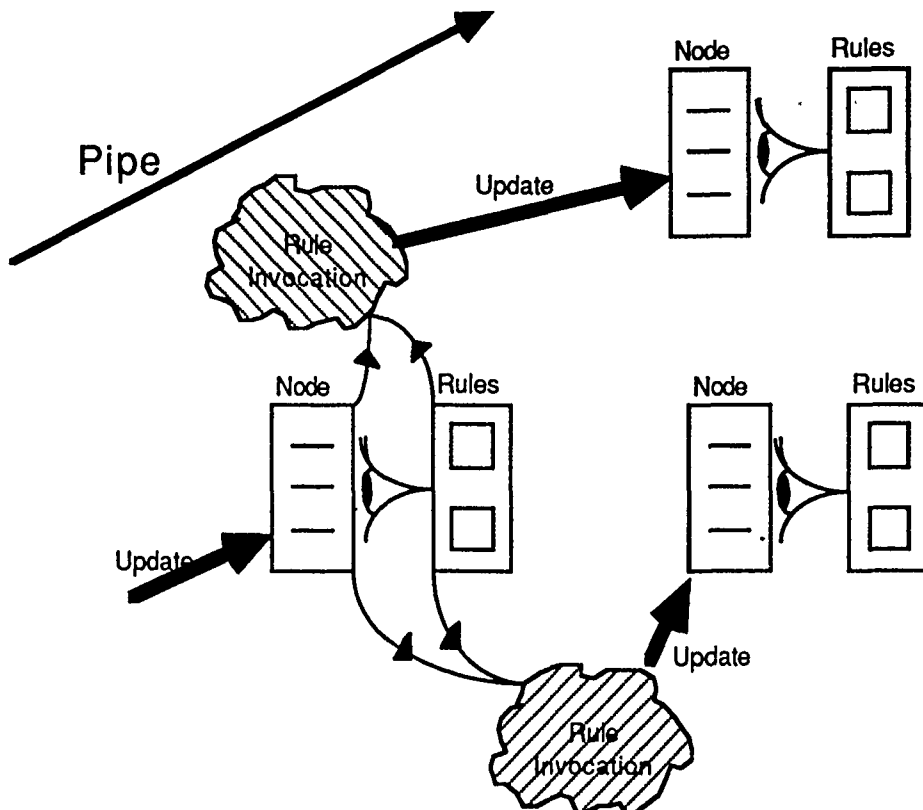


Fig. 4. Polygon's Execution model. An update to a Node triggers concurrent Rule invocations, which in turn update other Nodes. Pipes are formed as changes to the Blackboard flow from one Node to another.

3.3. How Polygon matches the problem domain

Polygon is not a general purpose programming language, other than in the *Turing complete* sense [Turing 36]. It is specialized to support one computational model and that computational model, itself, has limitations on its sphere of reasonable applicability. It has been designed with applications such as real-time signal understanding and data fusion in mind, though applications outside this domain are being investigated.

The structure of the problem domain is one that requires the representation of a large number of distinct entities in the solution space. For example the vocabulary of the problem domain is full of such things as aircraft, radar emitting platforms and radar track segments. Polygon provides a rich representation language in which these objects and specializations of them can be expressed. This allows the system to take full advantage of the mutual independence of any of the objects in the solution space to exploit parallelism.

3.4. How Polygon matches its target hardware

Polygon could, of course, run on any machine in principle. In practice, however, it has been designed with a particular kind of machine model in mind and has been optimized to take advantage of it. This class of target

machine, which was briefly described in Section 2.2.2, is exemplified by certain kinds of message-passing, distributed-memory multiprocessors. The grain size of the executable chunks in Poligon programs is designed to suit this model, i.e. each chunk represents, ideally, a few function calls. This makes it coarser grained than those systems that want to execute everything that can be in parallel, for instance data flow machines [Dennis 80], but it is a lot finer grained than most other concurrent blackboard systems, such as [Lesser 83] in which each processing element contains a complete blackboard system.

The target machine model, being of the distributed-memory, message-passing variety including essentially no capability to pass references, strongly discourages shared variables or mutable global data of any sort and encourages a message-passing style of programming. The Poligon language is one in which the programmer is given an abstract view of programming using the blackboard problem-solving model. The Poligon language has no construct for message sending at all, nor has it any primitives by which the user has access to the underlying architecture or topology. It is assumed to be the duty of the Poligon system or the target machine's operating system to look after such concerns. The Poligon compiler compiles its programs into the message passing primitives of the underlying system. This allows the efficient use of the underlying architecture, whilst still leaving the source program uncluttered by concrete details of the target architecture.

Poligon allows only global constants, but not variables, since these can be distributed at program load-time.

3.5. What we have learned

Truth comes out of error more easily than out of confusion. -

—Francis Bacon

Experiments with Poligon are by no means complete, but we have learned quite a bit so far. Some of these lessons are enumerated below.

- It is very hard to write any program which implements either a framework, such as Poligon or an application such as those which have been mounted on Poligon. This is due largely to asynchronous side effects. A system with better formal properties would be less error prone in this respect but might well make less efficient use of the hardware. These difficulties could also be caused by an insufficiency of mechanisms to control coherency in Poligon (see Section 3.1.1.)
- In order to produce a reliable program it is necessary to write code which makes no assumptions about anything that any other part of the system might be doing. Failure to do so results in brittle systems.
- In order to achieve a coherent solution it was found to be necessary to develop a number of programming methodologies. These will be covered in the same form as they were introduced in Section 3.1.1.

Node Level The creation of nodes is tricky. Because each element is likely to represent some real-world object, such as an aircraft, it is important either to provide a mechanism for resolving the conflict caused by multiple asynchronous requests to create an element that represents the same thing or to provide a mechanism for managing the creation of nodes. Poligon opts for the latter approach.

Slot Level The programmer should cause each node to have an idea of how to improve its own idea of the solution - to have *Goals*. In Poligon this is done at a fine grain, with each field of each element in the solution being able to have associated with it functions which enable it to evaluate itself. This state of affairs has been observed in a different manifestation at a larger grain size in [Corkill 83].

It was found that a good axiom for programming these systems is "Never throw away any data unless you are convinced that you have better data." This is the sort of behavior that is used in the evaluation functions mentioned above.

Rule Execution Poligon attempts to maintain the smallest critical sections possible. The original implementation of Poligon in fact had as its only atomic actions reading a field and writing a field. It was soon found that, in order to maintain consistency during rule execution, it had to be possible to read the values from a number of fields simultaneously - taking a snapshot without the subject moving. This, coupled with critical sections for the writing of collections of values, allows confidence that the picture that one sees when taking such a snapshot of a node is consistent, even if not necessarily the most up to date. It is important for a Poligon programmer to be aware that the node of which a snapshot has been taken may well be read from and written to by other rules asynchronously during the invocation of the rule taking the snapshot.

4. Experiments

In this section we describe, briefly, a series of experiments being performed by the Advanced Architectures Project [Rice 88] at Stanford University on the Poligon system and on Cage [Aiello 86] and Lamina [Delagi 86], other systems developed as part of the same project. However, these experiments will be discussed only in the context of the Poligon system.

It would be premature to quote any hard and fast performance figures here, since we still have much to do in order to understand the results that we are getting. The main purpose of reporting these experiments is to show

the lessons that have been learned both from performing the experiments and about the ways in which Poligon behaves.¹

4.1. The Problem

Each of the systems mentioned above has been used to implement an application called "Elint", a problem in the domain of real-time interpretation of passive radar signal data [Brown 86].

The problem is one of receiving reports from radar systems, abstracting these into hypothetical radar emitting aircraft and tracking them as they travel through the monitored airspace. These aircraft are themselves abstracted into clusters - perhaps formations - which are themselves tracked. The nature of the radar emissions from the aircraft are interpreted in order to determine the intentions and degree of threat of each of the clusters of emitters.

The Elint application has a number of characteristics which are of significance.

- The system must be able to deal with a continuous data stream. It is not acceptable to wait until all of the data has been read in and then figure out what is going on.
- The application domain is potentially very data parallel. The ability to reason about a large number of aircraft simultaneously is very important. What is more, the aircraft themselves, as objects in the solution space, are quite loosely coupled.
- The application is knowledge poor. This means that the experiments performed were geared primarily to evaluating the performance of these systems with respect to data parallelism, not knowledge parallelism.

4.2. The Purpose of the Experiments

I see no mention of God.

—Napoleon

I had no need of that hypothesis.

—Laplace

These experiments have five main objectives.

- 1 To investigate methods of achieving speed-up for expert systems applications by mounting them on parallel hardware architectures.²
- 2 To build a number of systems using different computational and problem-solving models and compare their relative performance and thus

¹Since this paper was written, many more experiments have been performed and their results published in [Delagi 88] and [Nii 88].

²"Expert Systems" are AI systems which attempt explicitly to encode the knowledge of human experts.

to deduce an appropriate course for future research. It is therefore imperative that, to the greatest degree possible, each of the systems should implement the same application and should perform the same experiments.

- 3 To perform experiments on individual systems specialized to investigate characteristics of each computational model, which might not be shown by the experiments mentioned above and which are not shared by other systems.
- 4 Having done the above, it should be possible to draw some conclusions about the amount of speed-up attainable given these architectures. This should help one to conclude whether these architectures are in fact appropriate and efficient for parallel implementation.
- 5 The implementation of the Elint system in Poligon was intentionally not tuned. This means that it was a copy of the original serial implementation modified only in so far as it was necessary in order to make it solve problems correctly in parallel. The intent was to achieve a reasonable measure of the performance of an average system that might be written by a Poligon user, as opposed to a very highly tuned version.

4.3. A Description of the Experiments Performed on Poligon

Deciding exactly which experiments to perform is difficult, since there are a very large number of variable factors in the system. Amongst these are; the implementation of the Elint system, the characteristics of the data sets used and numerous machine simulation parameters including processor and communications network performance. However, it was decided to freeze most of these and perform a number of experiments, having chosen "reasonable", justifiable values for the frozen parameters. We have, in fact, learned a lot from this process and this has helped us to design a better set of experiments, which are now being performed.

The primary variable factor for these experiments is the data set used to drive the experiment. This data set represents a simulated set of radar observations. These data sets are of finite length. The *length, number of simulated emitters and radar observation frequency over time*¹ are the main variable factors in the data sets.

To perform each of these experiments the simulated rate at which data arrived in the system was fixed at a value which was high enough to prevent data starvation when running the experiment on the largest reasonable processor grid. This meant that the speed-up for a grid of size N could be measured simply by dividing the time taken for the grid of size 1 by the time taken by the simulation of the N sized grid.²

¹Radar system reports per simulated time unit

²Performing experiments in this way was intended to give a base-line set of results of the same form as those derived from the CAOS system's implementation of Elint [Schoen 86] and of the Lamina implementation of *Airtrac*, another application [Nakano 87]. For the reasons mentioned in this section this might not be a good base-line for comparison.

It should be noted that these early experiments are open to some criticism as being unrealistic. They represent the speed-up for given programs under some fixed conditions. The conditions that are fixed may not be reasonable. For instance, if the program being run was merely a parallel implementation of *Quicksort* then these would be reasonable experiments. Unfortunately, because the implementations of Elint are intended to be real-time¹ systems it is not realistic to load the system in this way. The problem-solving behavior of the system is sensitive to machine load. Systems running with smaller numbers of processors will be more heavily loaded. They may, therefore, spend a lot of time queue thrashing.

For this reason it is now known that these experimental results should not be taken at face value. More satisfactory experiments have been devised, in which the experiment is run for a given number of processors with the data rate being varied until the latency of the output traces is constant over time. This means that the maximum sustainable data rate without increasing latency in the system's outputs is the preferred measure of the speed-up for these systems.

4.3.1. Experiment 1

The Fusion Plasma requires a temperature of 500 million degrees, but I forget whether that's Centigrade or Absolute.

—Overheard by Arthur H. Snell, Oak Ridge National Laboratory.

This experiment was intended to be a simple cross comparison experiment, performed by all of the systems. Its data set was a simple, and quite small one, which contained observations of sufficient variety to exercise all of the system's required behavior.

The speed-up figures produced showed a peak speed-up for the system of about 4.5X for sixty-four processors, with the speed-up trailing off quite sharply. This was disappointing.

One of the problems with this experiment was that the data set was varied in the frequency of input data for the system over time. It was sparse at the beginning, heavy in the middle and sparse at the end. This resulted in the system being data starved near the beginning of the simulation and then flooded in the middle.

Although such spikes in input data are entirely characteristic of real data, this extra variable factor was thought to be too difficult to factor out, in order to arrive at a realistic speed-up figure. If the system is lightly loaded then not much speed-up is needed. For this reason all subsequent experiments have been and will be performed on data sets that have a constant frequency of input data.

¹"Real-Time" is used here in the sense that the system must cope with an unbounded continuous stream of data, whilst delivering results reasonably promptly. It is not intended to refer to those real-time systems where guaranteed response times might be required.

The most important thing to conclude from this result is that we had much to learn about how to conduct these experiments.

4.3.2. Experiment 2

This experiment was designed to compensate for the variability found in the data set used in Experiment 1. The data set had a constant frequency for input data over time.

This experiment showed that the peak speed-up had increased to about 7X, which was reached after sixteen processors. This result was somewhat better than that from Experiment 1, supporting our hypothesis that the shape of the input data was affecting our results. Analysis of the simulator's instrumentation indicated that the limiting factor in the parallelism detected was probably a bottleneck on a particular node representing a cluster of emitters. It also showed that even if all bottlenecks were eliminated, so that all pipes were balanced, a major limiting factor in the performance of the system was that there wasn't enough parallelism at this grain size available in the data set for this system to exploit.

4.3.3. Experiment 3

This experiment was intended to determine how efficiently the simulated hardware architecture was being used and thus show where effort would best be expended to speed up the system if the application could not be changed structurally. To achieve this Experiment 2 (see Section 4.3.2) was repeated a number of times but for each iteration the simulated speed of the processor was varied. This gave speed-up figures for processor performances which were 2, 4 and 8 times the speed of the processor simulated in Experiment 2.¹ All of the speed-up figures produced were then normalized against the case of Experiment 2. A significant reduction in the speed-up of the system would have indicated that the increasing performance of the processor was swamping the communication hardware, thus indicating that time and effort would better be spent on improving communication performance.

It was found that the normalized speed-ups matched each other very closely. This is taken to indicate that, if such a machine were to be implemented for Polygon programs, effort spent on improving the processor's performance or in optimizing the program would probably be rewarded by close to linear speed-up.

4.3.4. Discussion of Experiments: What we have learned

Experience is the name everyone gives to their mistakes.

—Oscar Wilde, *"Lady Windermere's Fan"*

¹For each of these experiments the simulated input data rate was also increased so as to factor out this change.

As has already been mentioned the experiments on these systems are in their infancy. It is essential for the reader to note, therefore, that these results should be taken as nothing more than indication of where our research is leading us, rather than hard and fast statements about the performance of these systems.

We have, however, learned quite a bit in the execution of these experiments. The more important of these lessons are listed below.

- Getting useful speed-up out of these systems, at least given the current level of our understanding and methodologies, is very difficult. The speed-ups shown for the experiments mentioned in this section may, indeed, have been achievable by very careful coding on a uniprocessor. These difficulties are characterized mainly by the difficulty of implementing the program and debugging it and of combating serial components in the processing.
- Problem-Solving systems such as the ones mentioned in this paper are significantly more complex than those programs normally implemented to evaluate experimental parallel hardware. Our difficulty in getting results indicates that there is more to getting useful speed-up for real problems than there is to demonstrating speed-up for Quicksort programs such as [Deminet 82].
- The domain of Real-time systems is one in which the AI community in general and this project in particular has little experience. This has made implementation of these systems and the analysis of them difficult. The selection of a different field for research, outside that of real-time systems, would have alleviated this problem but would have removed the area of experimentation from an important area of application where it is believed that speed-up through parallelism is both necessary and feasible.
- Real-time systems present a set of problems for performance evaluation so great that it is difficult to formulate easily analyzable experiments and draw worthwhile conclusions from them. These problems are caused by; the need for continuous data, end effects when the data is bounded in extent, the difficulty of defining suitable performance measures and Heisenbergian effects i.e. changes in system load during speed-up measurement changing the speed-up itself.
- Investigation of the amount of "Knowledge Parallelism" has been limited by the relatively small amount of knowledge available in this area. New applications are being sought in which more knowledge is available. This has concentrated the investigation on the extraction of data parallelism from these systems.
- The data sets for the experiments mentioned above are limited in the amount of data parallelism that can be extracted from them. To add

to this problem the Poligon system is sufficiently difficult to simulate that experiments with significantly larger data sets are probably not feasible.

- The immediate conclusion that one is led to by these results is that a relatively simplistic implementation of a system can lead to speed-ups of the order of 10X. It seems to be possible to get higher speed-ups from such systems but, at least at present, only by very careful coding and very careful and thorough instrumentation of the running system so that bottlenecks can be eliminated.
- So far, it has not been possible to demonstrate overall speed-ups of more than ~8X using Poligon. The hypothesis that Poligon's implementation of Elint will be able to exploit data parallelism as larger data sets are used remains, as yet, untested, though tentative results from an implementation of Elint in Lamina (~54X) and Airtrac in Lamina [Nakano 87] (~80X) give cause for hope, indicating that with larger data sets there definitely is more parallelism to extract.

5. Conclusions

There is something fascinating about science. One gets such wholesale returns of conjecture out of such a trifling investment of fact.

—Mark Twain, "Life on the Mississippi"

This paper has introduced the problems associated with attempts to achieve speed-up through parallelism for *Problem-Solving* systems, systems developed in the *Artificial Intelligence* field. Numerous applications for such systems would benefit greatly from being sped-up considerably. Because of their irregular structure, such systems are shown to be difficult to speed up through well established means.

The Poligon [Rice 86] system was described. Poligon is an attempt to create a system which is able to encourage the decomposition of a particular class of Problem-Solving systems, known as *Blackboard Systems*, into a form, which can be efficiently executed by it on a distributed-memory, message-passing multiprocessor.

The Poligon system has been implemented and an application called "*Elint*" has been implemented using it. Lessons learned in the implementation of Poligon and the Elint application are detailed.

Experiments are now being performed on the Elint application, both for the implementation mentioned in Poligon and also for systems called Lamina [Delagi 86] and CAGE [Aiello 86]. Some preliminary experimental results are shown. Lessons learned from these experiments are described. Some of these are as mentioned below.

- It is very difficult to implement both frameworks for concurrent Problem-Solving and concurrent Problem-Solving systems themselves.

This is due largely to the difficulty of coping with asynchronous events, caused largely by these systems being MIMD systems.

- Real-time systems are difficult systems to calibrate for the purposes of experimentation to evaluate speed-up.
- Modest speed-up has been achieved (~8X). Indications of higher performance (~54X-80X) are thought possible through the exploitation more data parallelism [Nakano 87].
- The potential for the exploitation of knowledge Parallelism has not yet been investigated.
- If these results are supported by further work they would indicate that large amounts of parallelism at this grain size might not be easily achieved for this type of AI system. Thus, if there is not a lot of knowledge to apply, if there is not a lot of data parallelism available and if there are not many alternatives to explore in the application it may be that a software architecture optimized for a distributed-memory hardware architecture is not appropriate. This does not mean, however, that implementation techniques such as data copying and a message passing metaphor often used in distributed memory systems are not appropriate for a shared memory implementation, since they can help to avoid bottlenecks.

Report writing, like motor-car driving and love-making, is one of those activities which every Englishman thinks he can do well without instruction. The results are of course usually abominable.

—Tom Margerison, reviewing *Writing Technical Reports* by Bruce M. Cooper in the *Sunday Times*, 3 January 1965

6. Bibliography

- [Aiello 86] N. Aiello, "User-Directed Control of Parallelism; The CageSystem," Technical Report KSL-86-31, Heuristic Programming Project, Computer Science Department, Stanford University, 1986.
- [Brown 86] H. Brown, E. Schoen, B. A. Delagi, "An Experiment in Knowledge-Base Signal Understanding Using Parallel Architectures," Technical Report STAN-CS-86-1136, Heuristic Programming Project, Computer Science Department, Stanford University, 1986.
- [Byrd 87] G. T. Byrd, B. A. Delagi, "Considerations for Multiprocessor Topologies," Technical Report KSL-87-07, Heuristic Programming Project, Computer Science Department, Stanford University, 1987.

- [Clark 85] K. Clark and S. Gregory, "PARLOG: Parallel Programming in Logic," Technical Report Research Report DOC 84/4, Dept. of Computing, Imperial College of Science and Technology, London, 1985.
- [Clocksin 81] W. F. Clocksin and C. S. Mellish, "Programming in PROLOG," Springer-Verlag, Berlin 1981.
- [Corkill 83] D. D. Corkill and V. R. Lesser, "The Use of Meta-Level Control for Coordination in a Distributed Problem Solving Network," in *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, William Kaufmann Inc., Los Altos, 1983.
- [Delagi 86] B. A. Delagi, N. P. Saraiya, G. T. Byrd, "LAMINA: CARE Applications Interface," Technical Report KSL-86-67, Heuristic Programming Project, Computer Science Department, Stanford University, 1986; in *Proceedings of Third International Conference on Supercomputing*, pp. 12-21, Boston, MA, March 1988 International Supercomputing Institute.
- [Delagi 88] B. A. Delagi and N. P. Saraiya, "ELINT in LAMINA: Application of a Concurrent Object Language," Technical Report KSL-88-33, Heuristic Programming Project, Computer Science Department, Stanford University, 1988; SIGPLAN Notices, February 1989.
- [Deminet 82] J. Deminet, "Experience with Multiprocessor Algorithms," *IEEE Transactions. on Computers* C-31(4):278-287, April 1982.
- [Dennis 80] J. B. Dennis, "Data Flow Supercomputers," *IEEE Computer*, 48-56, November, 1980.
- [Flynn 72] M. Flynn, "Some Computer Organizations and their Effectiveness," *IEEE Transactions. on Computers*, C-21:948-960, 1972.
- [Gabriel 84] R. P. Gabriel and J. McCarthy, "Queue-based Multi-processing Lisp," in *Proceedings. of the ACM Symposium on Lisp and Functional Programming*, August 1984, pp. 25-44.
- [Gupta 86] A. Gupta, "Parallelism in Production Systems," Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, March, 1986.
- [Hillis 85] W. D. Hillis. *The Connection Machine*. MIT Press, Cambridge, MA, 1985.

- [Kuck 81] D. J. Kuck, R. H. Kuhn, D. A. Padua and M. Wolfe. "Dependence Graph and Compiler Optimizations," in *Proceedings of the Eighth ACM Symposium on Principles of Programming Languages*, ACM Press, New York, January 1981.
- [Kung 78] H. T. Kung and C. E. Leiserson, "Systolic Arrays (for VLSI)," in I. S. Duff and G. W. Stewart (Eds.), *Sparse Matrix Proceedings*, Society of Industrial and Applied Mathematics, Philadelphia, 1978, pp. 256-282.
- [Lee 85] G. Lee, C. P. Kruskal and D. J. Kuck, "An Empirical Study of the Automatic Restructuring on Nonnumerical Programs," *IEEE Transactions on Computers*, 1985.
- [Lesser 83] V R. Lesser and D.D. Corkill, "The Distributed Vehicle Monitoring Testbed: A Tool for the Investigation of Distributed Problem Solving Networks," *AI Magazine*, Fall 1983, pp. 15-33.
- [Nakano 87] R. T. Nakano, M. Minami, "Experiments with a Knowledge-Based System on a Multiprocessor," Technical Report KSL-87-61, Heuristic Programming Project, Computer Science Department, Stanford University, 1987; also in a shortened form in *Proceedings of Third International Conference on Supercomputing*, pp. 22-24, Boston, MA, March 1988 International Supercomputing Institute.
- [Nii 80] H. P. Nii, "An Introduction to Knowledge Engineering, Blackboard Model and AGE," Technical Report HPP-80-20, Heuristic Programming Project, Computer Science Department, Stanford University, March, 1980.
- [Nii 86] H. P. Nii, "Blackboard Systems," Technical Report KSL-86-18, Heuristic Programming Project, Computer Science Department, Stanford University, April, 1986; *AI Magazine*, 7-2 and 7-3, 1986.
- [Nii 88] H.P. Nii, N. Aiello, J.P. Rice, "Experiments on Cage and Poligon: Measuring the performance of Parallel Blackboard Systems," Technical Report KSL-88-66, Heuristic Programming Project, Computer Science Department, Stanford University, 1988; in *Distributed Artificial Intelligence II*. L. Gasser and M. N. Huhns (Eds.). Pitman Publishing Ltd. and Morgan Kaufmann, 1989.
- [Petard 38] H. Petard, "A Contribution to the Mathematical Theory of Big Game Hunting," *Americal Mathematical Monthly* 45:446, 1938.

- [Rice 86] J.P. Rice, "The Poligon User's Manual," Technical Report KSL-86-10, Heuristic Programming Project, Computer Science Department, Stanford University, 1986.
- [Rice 88] J.P. Rice, "The Advanced Architectures Project," Technical Report KSL-88-71, Heuristic Programming Project, Computer Science Department, Stanford University, 1988.
- [Schoen 86] E. Schoen, "The CAOS System," Technical Report KSL-86-22, Heuristic Programming Project, Computer Science Department, Stanford University, 1986.
- [Seifert 34] H. Seifert and W. Threlfall. *Lehrbuch der Topologie*, Academic Press, New York, 1934.
- [Seitz 85] C. L. Seitz, "The Cosmic Cube," Communications of the ACM, 28: 22-33, 1985
- [Steele 84] G L. Steele Jr., *Common Lisp the Language*, Digital Press, Burlington, MA, 1984.
- [Turing 36] A. M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," in *Proceedings of London Mathematical Society*, 2(42, 43): 230-265, 544-546, 1936.
- [Wilson 87] A. W. Wilson Jr., "Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors," in *Proceedings of the Fourteenth Symposium on Computer Architectures*, IEEE Computer Society Press, Washington, D.C., 1987, pp. 244-252..

The Elint Application on Poligon: The Architecture and Performance of a Concurrent Blackboard System

by
James Rice
(Rice@Sumex-Aim.Stanford.Edu)

**Knowledge Systems Laboratory
Stanford University
701 Welch Road, Bldg. C,
Palo Alto, CA 94304**

The author gratefully acknowledges the support of the following funding agencies for this project; DARPA/RADC, under contract F30602-85-C-0012; NASA, under contract number NCC 2-220; Boeing Computer Services, under contract number W-266875.

Abstract

In this paper we discuss Polygon, a skeletal system for the development of concurrent blackboard based applications, its architecture and the motivation for its design. A number of experiments have been performed in order to evaluate the performance of Polygon. Some of these are detailed and the results are shown. Lessons learned in the development of Polygon are given and conclusions about the performance of similar systems are drawn.

1. Introduction

It is often said that future AI applications will make significantly greater computational demands than the present generation. The Advanced Architectures Project of Stanford University's Heuristic Programming Project [Rice 88b] is investigating this issue, since it has as its objective achieving computational speed-up for expert systems through the use of parallel hardware and new, advanced software architectures. This requires the development of everything from designs for parallel hardware, which might be appropriate for the execution of future symbolic programs, through operating system and language concepts to problem-solving frameworks and eventually mounting applications on them in order to test the new designs.

Polygon [Rice 86] is one of the problem-solving frameworks developed as part of the Advanced Architectures Project. In Section 2, we discuss Polygon's architecture as a design for a high-performance, concurrent blackboard system aimed particularly at the problem domain of soft real-time problems, and what motivated this design. Section 3 discusses the applications mounted on the Polygon framework and experiments performed on the Polygon system to measure its performance. Section 4 presents the results of these experiments and an interpretation of them. We conclude in Section 5 with a number of the lessons we have learned in the process and pointers for future research.

2. The Polygon Architecture

In this section we briefly discuss the architecture of the Polygon system. A more detailed description of the design rationale for Polygon can be found in [Nii 88]. Because of space constraints, it will be assumed that the reader is conversant with the terminology of Blackboard Systems [Engelmore 88], though no deep knowledge will be assumed.

When we started the Advanced Architectures Project we had a hunch that the Blackboard problem-solving architecture [Nii 86] might offer a basis for the efficient exploitation of concurrent hardware. This was because the blackboard model appeared to have concurrency built into it. Why this is, in fact, not the case is explained in [Rice 88a]. The primary reasons why the blackboard model of a collection of simultaneously cooperating experts cannot develop the parallelism that one might expect is that the blackboard model itself assumes effectively infinite bandwidth with which the experts can see any part of the blackboard that might be of interest. It also assumes that experts do not get into one another's way whilst solving the problem. In practice a knowledge source can only see a small segment of the blackboard at any one time without degrading the performance of the system unacceptably. Similarly, the experts are dependent on one another, they must often wait for the results deduced by other agents and can be confused by updates being posted at unexpected times or in surprising orders. We are, however, unaware of a better architecture for concurrent problem-solving than that of Blackboard systems.

Although a number of other research efforts have looked at concurrent blackboard systems, these have concentrated primarily on either the aspects of distributed, concurrent problem-solving, such as [Lesser 83] or on coarse grained parallel systems, such as [Fennell 77], [Aiello 86] or [Ensor 85]. Polygon is a finer grained system than these, directed particularly at gaining speed-up through parallel execution.

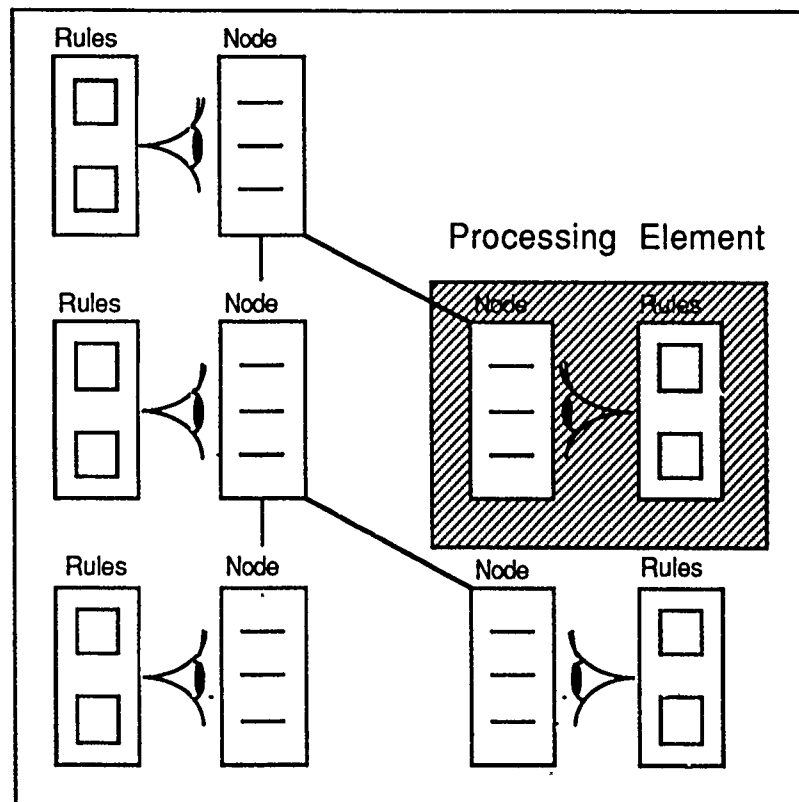


Figure 1. The Organization of the Polygon Blackboard. Rules are distributed over the network of processors and are attached to the blackboard nodes so that they can watch for modifications made to the slots in which they are interested.

The normal, serial implementations of the blackboard metaphor use a scheduling mechanism to cause one rule to fire after another. In parallel systems it is crucial that the programmer eliminate serial components, since this limits speed-up.¹ The main motivation of the Polygon system was to find a way to eliminate the serializing aspects of the blackboard model. We viewed this as doing the following:

- Eliminating the scheduling mechanism and finding ways to support concurrent rule activation all across the blackboard.
- Optimizing the design for distributed-memory, message-passing hardware, which should be able to deliver the best performance for large numbers of processors (of the order of hundreds to thousands.)
- Distributing the knowledge base over the blackboard so that there would be no serialization in the access to the blackboard from the executing knowledge.
- Designing the system so as to allow it to be highly compilable. It was clear from the outset that a considerable portion of the expense of existing AI systems is due to the fact that they are optimized for easy modification and debugging, rather than high run-

¹Speed-up can be viewed as the ratio of the system's speed using N processors to its speed using only one.

time performance. The resulting system, therefore, had to be designed so as to be able to be compiled efficiently yet still be intelligible and debuggable during the development cycle.

As these ideas progressed we developed the notion of a blackboard consisting of active nodes, tightly associated with the knowledge relevant to them.

A very simple scheme was developed for invoking the knowledge that had been distributed to the blackboard nodes: rules are activated as daemons as a result of modifications to the slots of a node (see Figure 1).

The distributed-memory hardware model, on which the Poligon system was to operate had the property that each processor was effectively a uniprocessor system. This meant that if we viewed the blackboard with a "Node as a Process/Processor" model then we would lose potential parallelism due to being able to execute only one piece of code (rule) at a time for any given node.

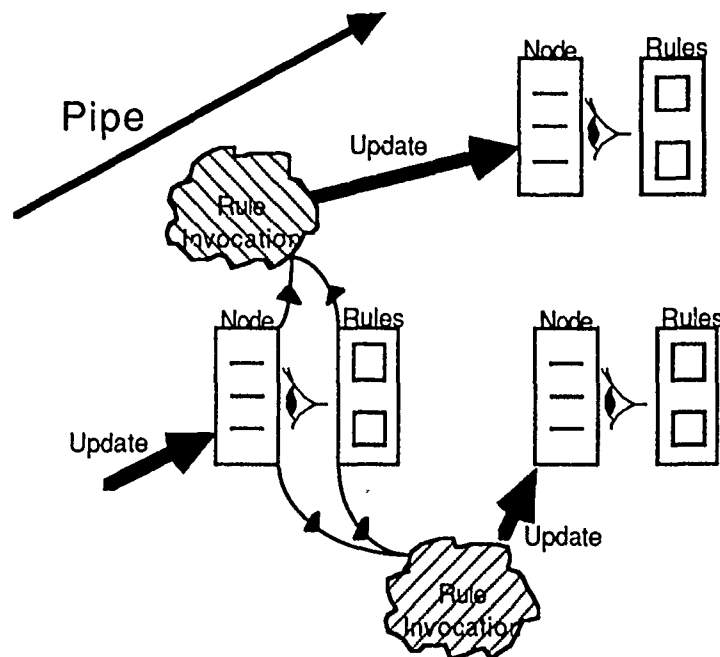


Figure 2. Updates to Poligon nodes cause concurrent rule execution, which themselves cause further updates. This implicitly forms pipes on the blackboard as data flows up or down the abstraction hierarchy.

What we needed, therefore, was a mechanism to allow the activation of multiple rules for any given blackboard node. This caused us to develop a model of Poligon which was as follows: "A blackboard node is a process on a processor, surrounded by a collection of processors able to service its requests to execute rules." It can easily be seen that this model is very close to a distributed object system model. This is by no means a coincidence. The underlying hardware system on which Poligon was implemented was a concurrent, distributed object oriented system [Delagi 86b].

The model expressed above is not without problems. In order to minimize the probability of a node being locked for a long period, which would delay remote access to it, as much

processing is done in the remote rule invocations as possible¹. This means that, when the rules execute, they have to do so in the context of a snap-shot of the solution state as it was when the rule was invoked (see Figure 2). Remote reads to other nodes, even the invoking node, are expensive and one cannot guarantee that things haven't changed by the time that the result of the read has been returned.

This led to the development of the idea of a Polygon node as being an agent capable of evaluating its own performance. Mechanisms had to be included so as to allow the system to be able to assess any request to modify its local state and to decide whether to perform the update, or what else to do instead, on the basis of its own view of how it is progressing towards its goal of solving the problem.

3. Experiments on Polygon

In this section we briefly describe the experiments performed on the Polygon system to date. Two applications have been mounted on Polygon: *Elint*, a soft real-time situation assessment problem and *ParAble*, a diagnostic application for particle accelerator beam-lines [Selig 87]. The experiments with the Elint application have now been completed, whereas those on the ParAble system are in their infancy, so only the Elint application will be considered here. A more detailed treatment of the Elint experiments can be found in [Nii 88].

The Elint application encodes knowledge used to interpret the radar emissions made by planes that are received by ground-based tracking stations distributed across the country. Because these tracking sites are passive devices, they can only detect the bearing and spectral characteristics of the radar emissions. Between them, it is their responsibility to deduce a position, course, identity and intention for any aircraft traveling through the monitored airspace. The Elint application simulates a central machine that integrates reports from these detection sites in order to achieve the overall goals mentioned.

The important characteristics of the Elint problem were:

- A continuous stream of input data.
- No *a priori* knowledge of the behavior or number of the aircraft being tracked.
- The need to emit periodic reports capturing the system's evolving view of the solution.

The Elint problem was chosen both because it was non-trivial and was in a class of problems, for which blackboard systems had already been used, and also because it was hoped that parallelism would be readily available. It was anticipated that parallelism could be extracted from the concurrent execution of knowledge on any given part of the solution space and from the potentially large number of independent elements in that solution space, i.e. aircraft.

The application was taken from a serial implementation and was not restructured so as to be better suited for parallel execution. The blackboard was, however, composed of three distinct layers in the abstraction hierarchy. Data flowing from one level to the next allowed pipes to be formed that were three stages long.

¹There are no user accessible locks in Polygon. Polygon nodes become locked (enter critical sections) during slot reads and updates, which are cheap operations. The Polygon architecture is such that deadlock will not happen as a result of system action, though the user can still write a program that will live-lock, e.g. two nodes each waiting for one node to update the other will wait for ever.

Perhaps the most important lesson that we learned from performing these experiments was to find a way to measure the relative performance of concurrent real-time systems. The best way that we found to do this was to pump data into the system at a given rate, which was under the control of the user, and examine the system's output over time. There is a measurable time between the time that data comes into the system and the time that any associated reports come out of the system. If this time difference increases on average over the course of a run then the system was not able to keep up with the rate at which data was being pumped into it. The experiment was then performed again with the data rate turned down until the report latency did not increase. This gave us a measure for the system's throughput, which we took to denote its peak performance.

The experiments that were performed were intended to measure a number of different aspects of the system's performance:

- The speed-up that the Polygon system could deliver.
- The peak throughput of the system.
- The ability of the system to exploit large knowledge bases.
- The granularity of the system.

Experiments to measure these are described in Section 4.

4. Experimental Results

The space available for this paper does not allow a full explanation of the experimental results, so the interested reader is again advised to refer to [Nii 88] for more details. It is hoped that the treatment here will be sufficient to give the gist of what we have learned.

It should be noted here that wherever reference is made to absolute times, these are measured in terms of the performance of the simulated hardware on which the Polygon system runs [Delagi 86a]. Each processing element of this machine is of about the performance of a TI Explorer™ II+ processor.¹

4.1. Measurement of Speed-up and Throughput

In this experiment two different data sets were used. One was designed to allow the Polygon system only to create one pipe in the solution space, the second allows Polygon to create four pipe-lines; it was four times as dense². The combination of these two results allows us to do the following:

- Measure the peak throughput for the larger data set.
- Determine the contribution to speed-up due simply to pipe-line parallelism.
- Compare the results from the two data sets so as to be able to get a measure of the ability of the system to exploit parallelism in the source data, i.e. data parallelism.

The results from the two data sets are shown in Figure 3.

¹Explorer is a trade mark of Texas Instruments Corporation.

²The small data set can be thought of as representing only one aircraft, the second had four. The data was, therefore, no more *complex*, there was just more of it.

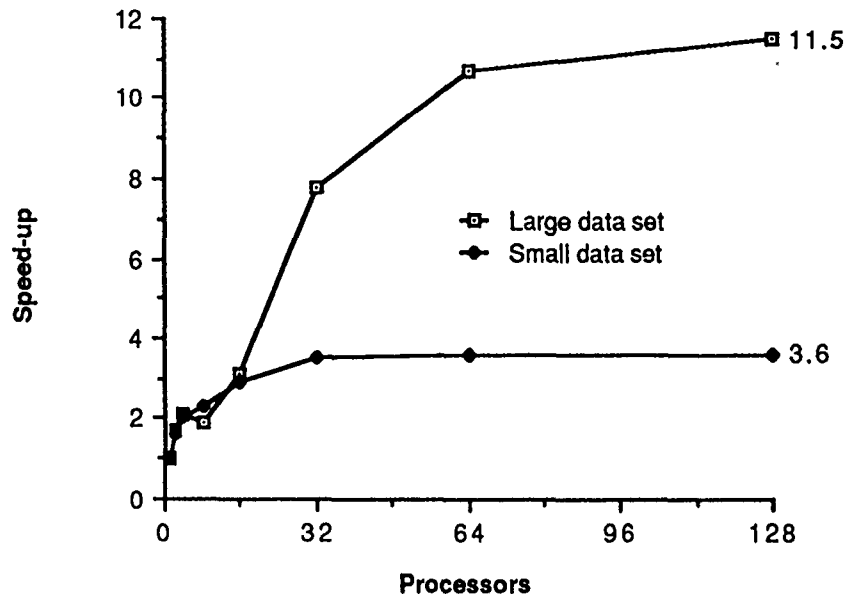


Figure 3. A graph showing the speed-ups derived from the large and small data sets plotted against the number of processors used.

In this experiment we learned the following:

- The peak speed-up shown in this application due to pipe-line parallelism was 3.6. This showed that although the length of the pipe was three, speed-up was greater than three because of the concurrent execution of rules by the different stages of the pipe.
- The peak throughputs measured from the two data sets were not significantly different. This indicates that Polygon was able to achieve an almost linear increase in speed-up as the problem size of the data set increased, an important result.
- The peak throughput for the system as measured from the larger data set was about 340 μ s per signal data record. Because of the linear increase in performance with data set complexity it is assumed that with more complex problems higher performance could be achieved. By comparison the Elint application, when coded to run in the AGE blackboard system took about 3.7 seconds to process each piece of signal data.

4.2. Measurement of Polygon's Ability to Exploit Large Knowledge Bases

In this experiment the Polygon system was tested using the small data set used above. The Polygon framework was modified so that, whenever a rule was invoked, N rules would be invoked, rather than just one. $N - 1$ of these rules had the special characteristic that they performed almost all of the processing required except for any blackboard modifying updates. This gave a measure of the system load if the knowledge base was N times larger, whilst still giving the right behavior for this application.

The results from this experiment are shown in Figure 4.

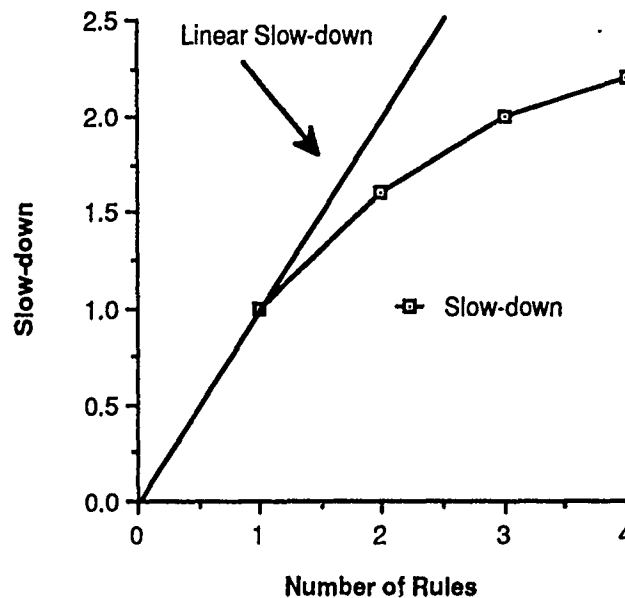


Figure 4. A graph showing application throughput slow-down plotted against the number of rules being fired for each rule-invoking event.

In this experiment, if the system were able to exploit parallelism in the knowledge base to the full, one would expect that the system would not slow down at all as new knowledge was added, i.e. the line shown in Figure 4 would be horizontal. If, on the other hand, the system bogged down completely as more knowledge was added one would expect that the result would be worse than linear slow-down, that is the plot would appear above the "linear slow-down" line. As can be seen easily from the graph, Polygon's performance was better than linear. In order to perform four times as much work it took only 2.2 times as long. This means that, as long as there are sufficient computational resources, the Polygon system delivers good performance for a knowledge base whose size is at least up to four times that of the Elint application.¹

4.3. Measurement of the Granularity of Polygon's Rules

In this experiment some of the internal mechanisms within Polygon were timed in order to get some empirical measure of the granularity of the system.

Within a blackboard system a number of mechanisms are of crucial importance to the performance of the system. Amongst these are slot reads, slot writes and rule invocation.²

In order to determine the costs of these operations they were performed repeatedly in a manner which allowed the individual costs to be measured with some precision.

The results of these experiments are as follows. It should be noted that all of these results neglect any communication overhead, so they are only representative for local operations.

- Slot reads take $1.36 + 0.94n$ μ s, where n is the number of slots being read at once, Polygon supports a form of multiple slot read operation.

¹In AGE, the Elint knowledge base was composed of about twenty knowledge sources, each having about three rules.

²Node creation is another important aspect, which was not measured in this experiment.

- Slot updates take $18 + 53.7n$ μ s, where n is the number of slots being written. Poligon allows arbitrary user code to be executed during the slot update operation, so this is a representative figure taken from the Elint application. This is for the case of no rules being associated with the slots being updated.
- The overhead cost of starting up a rule's execution is about 1ms per rule invoked.

A substantial part of the time taken performing these operations could be optimized considerably. For instance, a figure of about 500 μ s for rule invocation could relatively easily be achieved in a real system and more than this improvement could be expected for a system which allowed specialized microcode or similar efficiency tuning. This shows that there is a lower bound to the granularity that the user can expect to achieve. For computations taking less than a few milliseconds it may not be worth starting up a rule to perform the computation, the cost of parallel execution would be in excess of the serial execution time.

5. What We have Learned

We have learned a number of lessons from this project, some of which were counter to our intuitions.

- Our intuition told us that programming a concurrent blackboard system would not be too hard because of the assumed implicit asynchrony in serial blackboard systems. We found this not to be the case. We found the programming task to be difficult and, we believe, a reconceptualization of existing problems will be required in order to port them for efficient parallel execution. The difficulty of implementation of applications is due largely to the divergence of implementations of serial blackboard systems from the pure blackboard model in order to make implementation and programming more manageable as was mentioned in Section 2 and is covered more thoroughly in [Rice 88a].
- We found that the Poligon system and architecture itself performed fairly well. Although programming the system was not trivial, the Poligon system provided a useful abstraction model that allowed the development of an application in a blackboard-like manner that still gave the correct answers and acceptable performance.
- We had thought that parallelism in the knowledge base would be crucial to the achievement of high performance. In the applications that we used, knowledge proved to be sparse and the pipe-line parallelism that resulted from it delivered only a factor of three in speed-up. The small amount of speed-up from pipe-line parallelism was due to the short length of the pipes, the lack of applicable knowledge and the difficulty in balancing the pipes. Most of the parallelism seen in the applications implemented in the Advanced Architectures Project was derived from the data, not the code. The limit to the length of the pipes derived from the application was not one that resulted from the structure of the problem itself, but rather came from the fact that the application was reimplemented for Poligon from the AGE implementation, not reformulated.
- When we started the project our intuition told us that the significantly greater cost of communication relative to computation would bias the programmer in favor of doing as much as possible locally before a message was sent. It turned out that doing this increased the granularity of the system and restricted parallelism. We found that, although communication is expensive, as long as data keeps flowing along a pipe the price that is paid is in latency, not in speed-up. The fact that processes are not held up by communication is a result of the non-blocking message sending ability of the hardware. Thus, fine-grained systems are likely to be significant for achieving good performance from large multiprocessors but the increased latency due to distributing the work could have an adverse effect on real-time performance.
- We learned that simulation of multiprocessors is expensive. A number of projects are interested simply in the difficulties caused by the asynchronous behavior of concurrent

systems. Such projects are able to use a simple model for their implementations on existing hardware. We, on the other hand, wanted to measure the performance of our software on the hardware we were developing precisely in order to refine both our hardware and software designs. This is computationally a very expensive task and has proved to be a major limiting factor on the work that we have done. Having said this, however, it should be noted that we are confident that we have achieved better results and have gained deeper insights than we would have done if we had concentrated on building real hardware.

- Resource allocation was found to be a significant factor in delivering high performance. The fact that blackboard nodes are often very long-lived means that an even load balance can easily be disrupted by a few busy nodes. In the experiments reported here the allocation of processes to processors was done randomly. Other experiments in the Advanced Architectures project have shown that, compared to the ideal, perfect load balanced situation,¹ even with very careful site allocation the Elint application lost 30% in efficiency and delivered 30% less speed-up than in the perfectly load balanced case. This could not be recovered through the use of more processors [Delagi 88].

6. Conclusions

In this paper we have described Poligon, a blackboard framework designed to operate on distributed-memory multiprocessors. We have described experiments performed on it, shown the results and discussed the conclusions that can be drawn from them and mentioned some lessons that were learned along the way.

We have shown that the Poligon system can deliver a speed-up for the Elint application of nearly twelve, with near linear speed-up gain with increasing problem complexity. We have also shown significantly better than linear slow-down as a result of increasing knowledge base complexity. We are confident, therefore, that given a larger problem Poligon could deliver significantly more speed-up than this.

From our work we can conclude that data parallelism is likely to be the most important source of parallelism in the foreseeable future, at least until truly huge knowledge bases are developed. This requires that concurrent problem-solving systems should be not only able to exploit data parallelism but be able to do so in a manner which allows the rapid development, easy maintenance and modification of knowledge bases and encourages the development of software that is not brittle when knowledge is added or removed or when the system meets circumstances that were not anticipated by the programmer. Poligon is a possible first step in this direction.

7. Bibliography

- [Aiello 86] Nelleke Aiello. *User-Directed Control of Parallelism: The CAGE System*. Technical Report KSL-86-31, Knowledge Systems Laboratory, Computer Science Department, Stanford University, April 1986.
- [Delagi 86a] Bruce Delagi. *CARE Users Manual*. Technical Report KSL-86-36, Knowledge Systems Laboratory, Computer Science Department, Stanford University, 1986.
- [Delagi 86b] Bruce A Delagi, Nakul P. Saraiya, Gregory T. Byrd. *LAMINA: CARE Applications Interface*. Technical Report KSL-86-76, Knowl-

¹This was possible to measure because of being able to "cheat" in the processor allocation for the simulator, assuming global knowledge.

- edge Systems Laboratory, Computer Science Department, Stanford University, 1986.
- [Delagi 88] Bruce A. Delagi and Nakul P. Saraiya. *ELINT in LAMINA: Application of a Concurrent Object Language*. Technical Report KSL-88-33, Heuristic Programming Project, Computer Science Department, Stanford University, 1988.
- [Engelmore 88] Robert Engelmore and Tony Morgan (eds.) *Blackboard Systems*. Addison-Wesley Publishing Company Inc., Menlo Park 1988.
- [Ensor 85] J. Robert Ensor and John D. Gabbe. *Transactional Blackboards*. Proceedings of the 9th International Joint Conference on Artificial Intelligence: 340-344, 1985.
- [Fennell 77] Richard D. Fennell and Victor R. Lesser. *Parallelism in AI Problem Solving: A case Study of Hearsay-II*. IEEE Transactions on Computers: 98-111, February, 1977.
- [Lesser 83] Victor R. Lesser and Daniel D. Corkill. *The Distributed Vehicle Monitoring Testbed: A Tools for the Investigation of Distributed Problem Solving Networks*. The AI Magazine, Fall:15-33, 1983.
- [Nii 86] H. Penny Nii. *Blackboard Systems*. Technical Report KSL-86-18, Knowledge Systems Laboratory, Computer Science Department, Stanford University, April 1986. Also in AI Magazine, vol. 7-2 and vol. 7-3, 1986.
- [Nii 88] H. Penny Nii, Nelleke Aiello and James Rice. Experiments on Cage and Polygon: Measuring the Performance of Parallel Blackboard Systems. Technical Report KSL-88-66, Knowledge Systems Laboratory, Computer Science Department, Stanford University, October 1988.
- [Rice 86] James Rice. *The Polygon User's Manual*. Knowledge Systems Laboratory, Computer Science Department, Stanford University, 1986.
- [Rice 88a] James Rice. *Problems with Problem-Solving in Polygon: The Polygon System*. Technical Report KSL-88-04, Knowledge Systems Laboratory, Computer Science Department, Stanford University, January 1988. Also in Proceedings of Third International Conference on Supercomputing, May 1988.
- [Rice 88b] James Rice. *The Advanced Architectures Project*. Technical Report KSL-88-71, Knowledge Systems Laboratory, Computer Science Department, Stanford University, January 1989.
- [Selig 87] Lawrence J. Selig. An Expert System using Numerical Simulation and Optimization to find Particle Beam Line Errors. Technical Report KSL-87-36, Heuristic Programming Project, Computer Science Department, Stanford University, 1987.

The Advanced Architectures Project

by
James Rice
(Rice@Sumex-Aim.Stanford.Edu)

**Knowledge Systems Laboratory
Stanford University
701 Welch Road
Palo Alto, CA 94304**

The author gratefully acknowledges the support of the following funding agencies for this project; DARPA/RADC, under contract F30602-85-C-0012; NASA, under contract number NCC 2-220; Boeing Computer Services, under contract number W-266875.

Abstract

The Advanced Architectures Project at Stanford University's Knowledge Systems Laboratory seeks to gain higher performance for expert system applications through the design of new, innovative software and hardware architectures. This research is concentrating particularly on the use of parallel machines to gain speed-up and the design of the software to exploit emergent parallel hardware architectures. The project is described, detailing its goals and the work performed in the pursuance of those goals. A brief description is given of each of the components of the project and a complete bibliography is given of the publications produced by the project.

1. Introduction

The Advanced Architectures Project¹ is a project which has been running for a number of years at Stanford University's Knowledge Systems Laboratory. The project is large and has a number of components, which have been documented at length. The project as a whole has never been drawn together in one document, so the purpose of this paper is to describe the Architectures project, and to give a taste of the individual sub-projects, which have kept us so busy for so long. A large number of publications have emerged from the project so, we will also take this opportunity of giving a full bibliography of the work we have done so that the interested reader can follow up on any intriguing topics.

The AAP straddles a number of areas of research and because of this does not fall easily into the sphere of interest of any one camp. A certain amount of work has been done on the parallelizing of expert systems, most notably by Gupta [Gupta 86]. Similarly, there are machines on the market that are similar in some respects to the machines that we have been designing, most notably the Ametek machine. This paper does not in any significant way relate the work on the AAP to other work in these fields. This is done copiously in the papers cited below. The reader should keep in mind, however, that there have not been, to the best of our knowledge, any projects comparable to the AAP and so it is often very hard to find a reasonable point of comparison. For instance, considerable effort is being spent on concurrent image recognition on massively parallel machines such as the Connection MachineTM.²

At this point we should make a brief disclaimer. The subject matter for the Advanced Architectures project is complex and not in the main stream of experience of the AI community. We are concerned lest the conclusions we have drawn from our work should be misinterpreted, trivialized or over generalized. It is not realistic to try to do justice to the full lessons that can be learned from our extensive quantitative experiments in an article as short as this. For this reason we do not attempt to draw any great conclusions here. The reader is strongly encouraged to read any of the fifth odd papers that are cited herein so as to gain a deeper understanding of our work and the lessons we have learned. All of these publications are available from the KSL and many are also published in other fora. The reader is, therefore, encouraged to treat this more as a *concept* or *research in progress* paper than as one that has scientific goals.

¹The project's real name is in fact "Expert Systems on Multi-processor Architectures", but all of the project members have always felt more comfortable calling it either the "Advanced Architectures" or the "Architectures" project, for short. This seems to capture better the fact that new architectures for both hardware and software will be needed in order to meet the project's goals.

²Connection Machine is a trade mark of Thinking Machines Corp.

1.1. Project Goals

The project's primary goal is to find ways to increase the performance of expert systems through the use of the new, emergent, parallel hardware designs.

The number of possible implementation strategies for such a project is huge. One has only to look at the large number of different hardware designs that are emerging and at the number of different problem-solving methods to see how combinatorial the problem would be if we endeavored to investigate all of the reasonable and plausible combinations of architectures. It was decided, therefore, that we could learn a great deal simply from making a commitment to one, or at least a small number of different options at each point in the system's make-up. We thus decided to take a "vertical slice" through the space of possible solutions. Clearly we did not intend to investigate any options that seemed non-useful, so we knew from the outset that, although we could not prove that we had the best design to meet our goals, our design would nevertheless be at least a plausible architecture for a future computational environment.

We viewed the task of implementing concurrent expert systems as being one which was split into a number of implementation layers. If we could achieve speed-up at each one of these layers, then we could hope for a substantial overall performance improvement compared to existing AI systems. Our model of the layers into which the project could be split is shown in Figure 1.

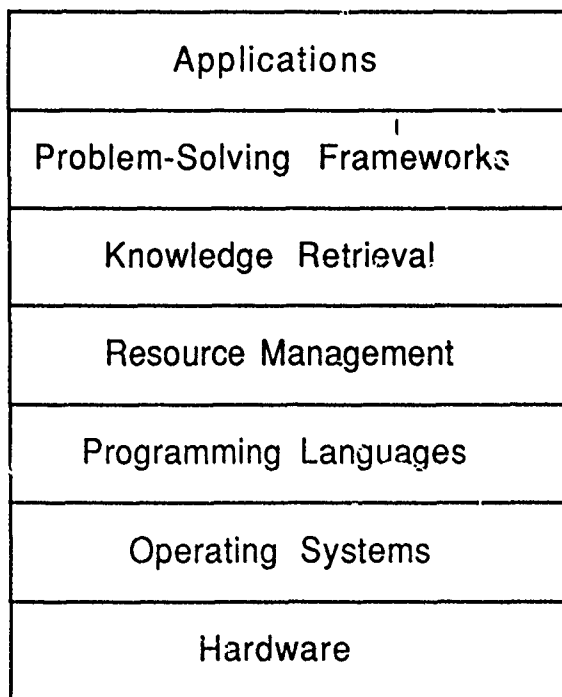


Figure 1. The layers of system implementation through which we hoped to achieve computational speed-up in the Architectures Project.

It was originally anticipated that the needs of the applications would drive the development of the problem-solving frameworks and so on down through the implementation hierarchy shown in Figure 1 until eventually the hardware would be designed under the constraints passed down from above. In practice, however, this did not happen. Because of the difficulty of finding and mounting an application suitable to our needs and the early availability of personnel interested in the hardware design aspect, the hardware design went ahead

more rapidly than the other layers. This resulted in our designs being more hardware driven than application driven. This is not necessarily a bad thing, since an entirely top-down design process could easily have resulted in low-level system requirements which were not implementable.

As well as the thrust of the project coming from the bottom rather than the top, the levels of abstraction actually implemented differed significantly from those shown in Figure 1. Figure 2 gives a more realistic representation of what we actually did, as opposed to what we intended to do.

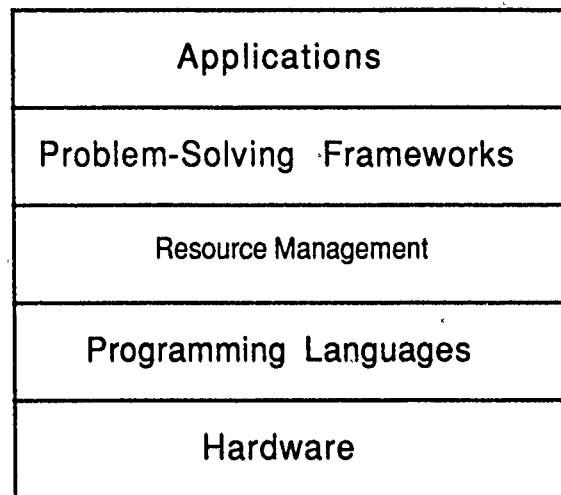


Figure 2. The layers that have actually been tackled in the Architectures project. Resource Management is shown in small type because it is a recent addition and most of our work has been done without the help of this layer.

The Knowledge Systems Laboratory has considerably more expertise in software, than in hardware department. We decided early on not to build any hardware - there are many other research groups that could do this better than we. We decided, therefore, to simulate our hardware. This would allow us to modify our software and hardware designs easily and allow us to extract the maximum insight with the minimum effort.

The rest of this paper is split into sections which reflect the major layers shown in Figure 2. In each of these sections the work of the relevant sub-projects will be discussed. Because of the bottom-up thrust of the project the project's components will be discussed in a bottom-up order. This will also reduce the number of forward references made, since discussion of the higher layers will inevitably have to refer to the substrates on which they are implemented.

1.2. Personnel

This project has employed a large number of people over the years and it seems appropriate to name them all here, since otherwise they might only appear as authors referenced in the bibliography.

Ed Feigenbaum, Bob Engelmores, Penny Nii, Bruce Delagi, Harold Brown, Hiroshi Okuno, John Delaney, Byron Davies, Hirotoshi Maegawa, Nelleke Aiello, James Rice, Nakul Saraiya, Sayuri Nishimura, Eric Schoen, Greg Byrd, Max Hailperin, Russel Nakano, Masafumi Minami, Chris Rogers, Alan Noble, Jean-Christophe Bandini, Manu Thapar, Pandu Nayak, Jerry Yan and Sam Hahn.

2. Hardware

As was mentioned above, hardware design led the way in the Architectures project. In this section we discuss a little bit of the motivation for the hardware designs and briefly describe both the current generation of hardware designs on which we are working and the simulator we are using.

2.1. Simple and Helios

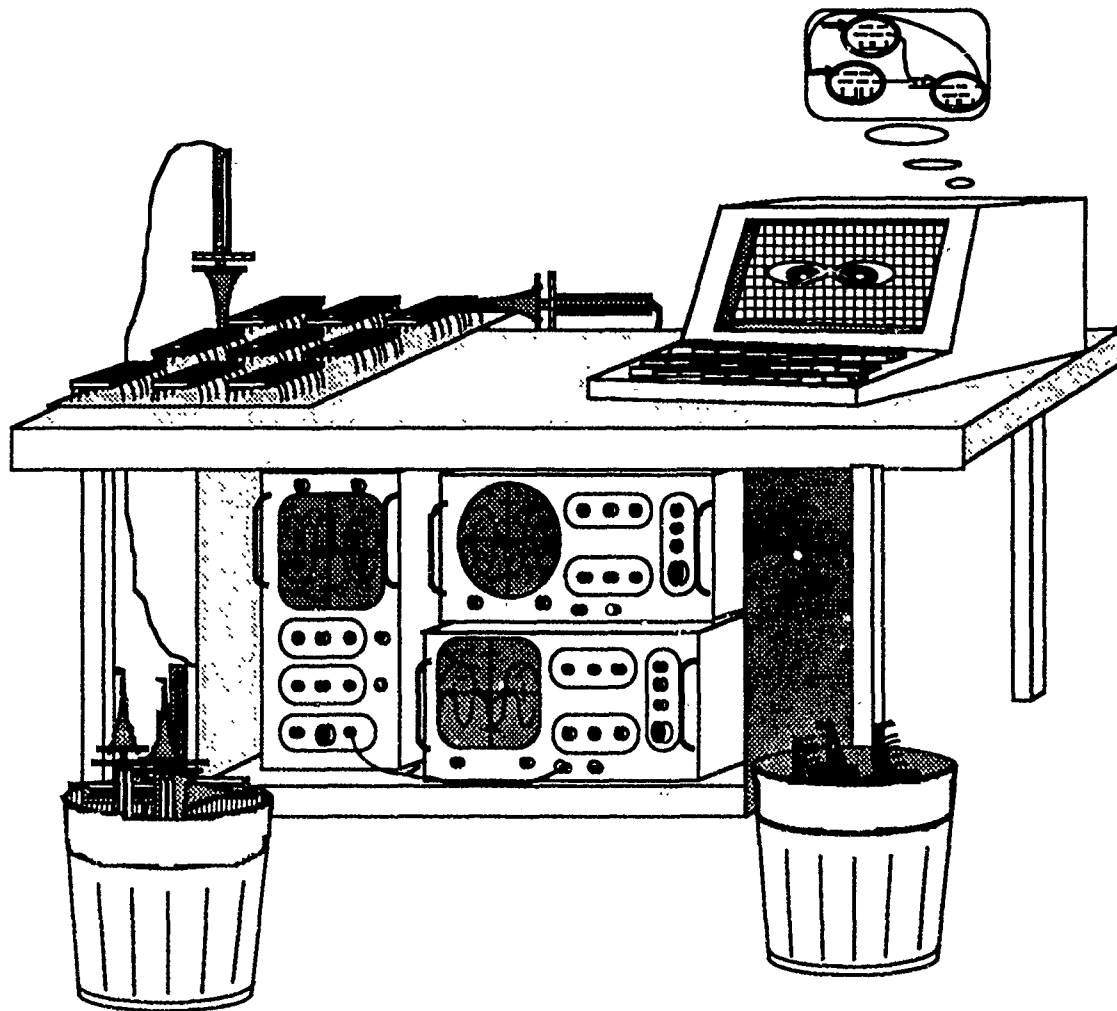


Figure 3. The Simple system provides a toolkit from which to build circuits to be simulated, a collection of probes to connect to the circuit and a set of instruments to connect to the probes.

The hub of all of the work done on the Architectures Project has been the circuit simulator, upon which everything else is built. This simulator is called *Simple*. It is an event-driven simulator, designed to allow the user to design and specialize digital circuits in a simple and modular way, using a circuit design tool called Helios. A sophisticated set of instrument tools allow the user to design and specialize simulated probes which can be connected to the circuit while it is running. This allows the connection of a number of instruments to the probes that permit the user to see the behavior of the circuit as it operates without interfering with the behavior of the system. We like to view this model as one of a laboratory workbench equipped with collections of instruments, probes and circuit building compo-

nents from which the user can build systems and on which the user can perform quantitative experiments (see Figure 3). More information on this topic can be found in [Delagi 86b] and [Delagi 87].

It was found early on that simulations of the sort we wanted to do would be computationally very expensive. An attempt was made, therefore, to parallelize the simulator itself in an attempt to bring down the times taken for the simulations, which often exceeded one day in duration. This resulted in *AIDE*, a distributed version of Simple [Saraiya 86]. Unfortunately, we were unable to achieve any speed-up at all for our simulations, largely because of the communication bandwidth and latency associated with communicating between the multiple Symbolics machines we were using via an Ethernet and because the simulator, being event-driven required frequent synchronization on the event queue, which serialized the processing.

2.2. CARE

The Simple simulator mentioned above was used to design and build what we refer to as the CARE¹ machine and simulation system [Delagi 88a] (see Figure 4). The CARE machine is that simulated machine on which all of the experiments mentioned below have been performed. The machine's design has a few key features which are worthy of note:

- Dynamic cut-through routing, in order to optimize network throughput [Byrd 87c].
- Toroidal topology [Byrd 87b].
- Non-blocking message sending, so as to encourage pipe-line processing.
- Communications network with alternative paths between points, so as to reduce communications problems due to busy communication paths.
- A separate communications controller, in order to support operating system functions and to implement the non-blocking send functionality mentioned above.

The work on the CARE sub-project has focussed mainly on the design of inter-processor communication networks, as is appropriate. This has meant that we have been able to ignore the instruction level behavior of the processors themselves. The application programs that we run are merely timed as they run between the points at which code fragments cause communication between processors. Being able to avoid doing register level simulation of the processors themselves has allowed us to execute much more complex and realistic programs on our simulated machines. We have therefore traded accuracy in our processor simulation - assuming that the processing elements will behave much like existing Lisp Machine processors - in favor of greater realism in terms of the system's performance under the load of real programs.

A number of aspects of system design have not been addressed in detail and the simulations do not take these into account. Most significant among these, perhaps are the fact that memory usage, code distribution and garbage collection are not simulated.

The CARE/Simple simulator system is perhaps the most valuable tangible product of the Architectures project. It is now being used in a number of research departments, both corporate and academic, outside Stanford. Like all Architectures Project software, it is in the public domain. CARE/Simple will soon be available running under Common Lisp, CLUE and X11 on a number of different platforms.

¹The expansion for this acronym seems to have been lost somewhere in the wash. We think that it has something to do with the words *Concurrent* and *Array*.

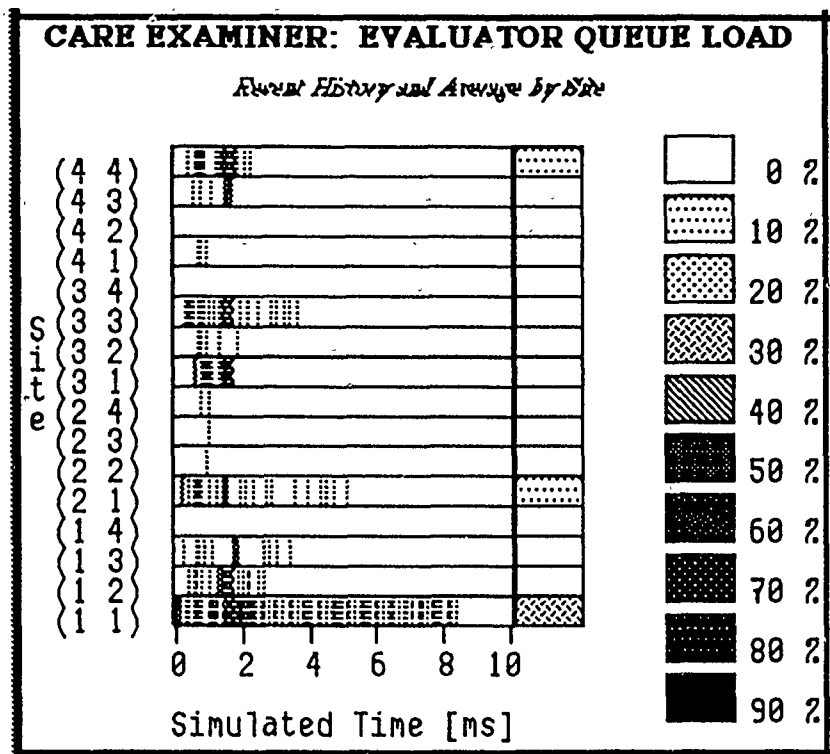


Figure 4. An example piece of instrumentation from the CARE system. This one shows the lengths of the task queues on the different processors plotted against time.

3. Operating Systems and Languages

A considerable amount of effort has been spent on the project in working at the operating system level of abstraction. Curiously, we have written no operating systems. The CARE machine itself features a dual processor for each processing element. This allows much of the work of the operating system, particularly inter-processor communication, to be done by a dedicated processor in parallel with the execution of user code. The behavior of this communication processor is coded directly into the simulated hardware.

Amongst the work that has been done in this area has been work on concurrent object-oriented systems, concurrent Lisp dialects, programming models and resource allocation.

3.1. CAREL

CAREL [Davies 86] was one of the first programs written to run on the CARE simulated machine. It was an early attempt to find a Lisp language interface to the distributed-memory hardware provided by CARE. It took as its basis Scheme [Abelson 83] and QLisp [Gabriel 84] and included primitives to allow remote function calls and remote consing. It was quickly found that, because of the cost of process creation, it was desirable to make the best use of any processes that were spawned. There was therefore a need to store application dependent data in non-ephemeral spawned processes. State of this type was implemented in CAREL as writable closure variables. These process closures could be used as elements in pipe-line computations or to represent mutable communicating program objects, for instance to represent real-world objects with state. State, as encapsulated in communicating objects, and the idea of pipe-line parallelism have been pivotal in the design of the other systems developed on the Architectures project.

The CAREL project was used mostly as a feasibility study and was soon discontinued.

3.2. CAOS

The first implementation of the Elint application, described further in Section 5.1, was made without the benefit of any problem-solving framework. It was anticipated that the application could be easily mounted almost directly on the CARE machine and some experiments could be run quickly, which would allow us to learn some important lessons early in the project.

In order to mount the application, a distributed object-oriented system was implemented. This was done because the CARE system did not, at the time, come with its own "preferred" object system. The system that was implemented was called CAOS [Schoen 86], a Concurrent Asynchronous Object-oriented System. It was implemented using the Flavors system supported by the Lisp machines used by the project. It had a number of key features:

- Each CAOS object was potentially a multiprocess object, though executing on a single processor, having at least one stack group associated with each CAOS object.
- CAOS objects were intentionally large grained. This was because it was anticipated that the communications network would be the resource most competed for, thus encouraging the programmer to perform a lot of computation in order to reduce the number or size of messages sent.
- Message-passing was used as the metaphor for communication in the language extensions provided by CAOS.
- A large number of different message sending primitives were defined, including non-blocking sends that did not require a reply from the target of the message, sends that returned futures to the values returned by the targets and send operations which blocked immediately in order to wait for a reply from their targets.

The CAOS system proved to be too expensive to use for our future experiments. Contrary to our intuition, the communications network proved to be the least loaded of the CARE machine's resources during our experiments on CAOS. The computational expense of supporting its complex object model caused the granularity of the resultant computations to be too large.

3.3. LAMINA

Lamina [Delagi 86a] is the object system that was designed after the lessons were learned from the CAOS experiments. It was originally intended to provide a very small, light-weight layer on top of the CARE machine so that distributed object-oriented programs could be implemented efficiently. A significant part of the motivation for the design of Lamina was the desire to reduce the overhead suffered by the CAOS system in terms of associating large stack groups with each of the CAOS objects. Lamina introduced the idea of objects with restartable, rather than resumable code segments, which do not require stacks to preserve their state when they are not running. Since its first appearance Lamina has been developed extensively and, although still small and light-weight it now provides a platform for the development of computational models for functional and shared-variable as well as object-oriented programming.

Lamina has been used to implement a number of programs, both for direct implementations of the two real-time expert systems being investigated (see Section 5), AirTrac and Elint, and also a number of numerical programs. Lamina is now the preferred core programming

system for the CARE machine and applications in Lamina have consistently shown the highest performance of all programs running on the CARE machine.

3.4. Inter-Processor and Inter-Process Communication

A considerable amount of work has been performed on the investigation of different mechanisms for inter-processor and inter-process communication. For distributed-memory machines we believe that the efficient distribution of work for large applications is crucially linked to the efficient implementation of multicast communication [Byrd 87a], [Byrd 88b]. Although the principal thrust of the project has been towards the development of distributed-memory hardware, the fact that the CARE simulator can also simulate shared-memory machines has allowed the investigation of the relative performance of these two distinct classes of machines and the relative performance and appropriateness of shared-variable and message-passing/object-oriented programming models [Byrd 88a]. Current work is focussing on the design of hardware that might provide efficient support for both the shared-variable and the message-passing programming models [Byrd 89].

3.5. Load-Balancing

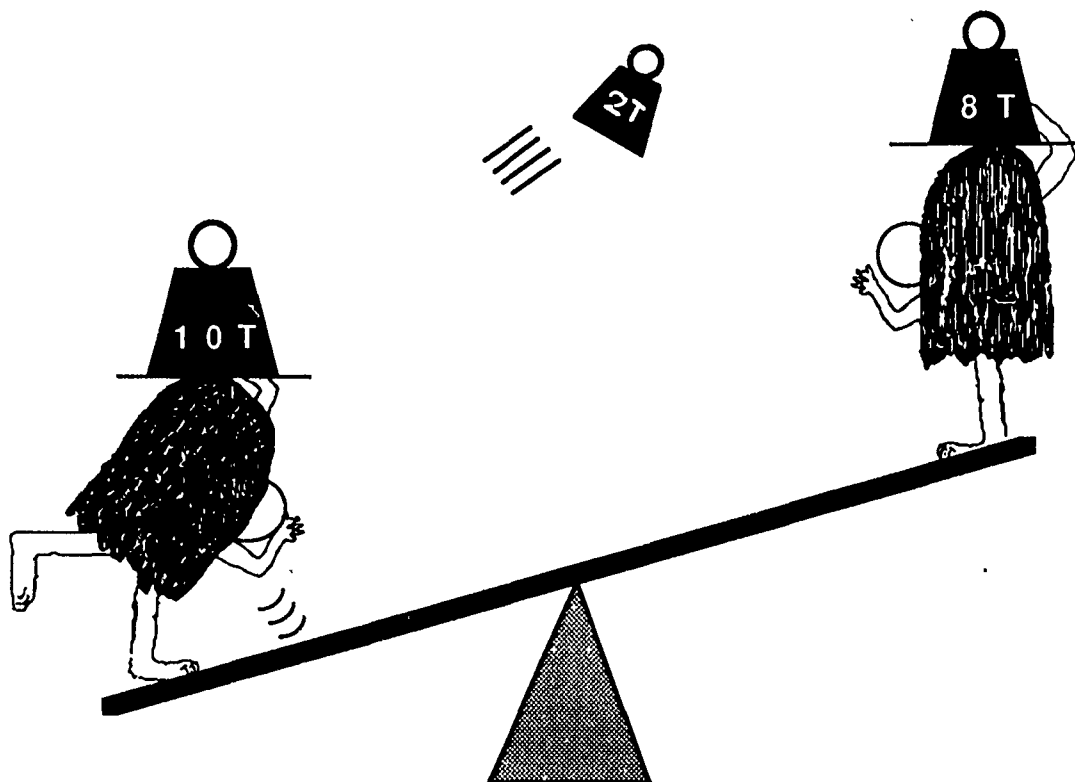


Figure 5. Load-balancing.

We have started to examine load-balancing problems (see Figure 5) within the context of the AAP "vertical slice." [Hailperin 88] In particular, this work is focusing on a load-balancing method intended to migrate Lamina objects in large (thousands of processing elements) CARE multicomputers in order to improve the performance of soft-real-time signal-interpretation systems such as Elint and AirTrac (see Section 5).

Without load balancing, only a lightly-loaded multicomputer, which has cause to create processes dynamically, can in general achieve real-time performance. The focus of the work is on how to achieve global load balancing, which would be an attractive solution to

this problem, as it would allow the effective use of massively-parallel ensemble architectures for larger soft-real-time problems.

The challenge is to replace quick global communication, which is impractical in a massively-parallel system, with statistical techniques. In this vein, a novel approach to decentralized load balancing is being investigated based on statistical time-series analysis. Each processing element estimates the system-wide average load using information about past loads of individual sites and attempts to equal that average. This estimation process is practical because the soft-real-time systems in which we are interested naturally exhibit loads that are periodic, in a statistical sense akin to seasonality in econometrics.

A load-balancing system for Lamina/CARE has been designed using this load-characterization technique, and its implementation and experimentation with it in the context of the ELINT and AIRTRAC applications have begun.

3.6. Concurrent and High Performance Lisp

In an attempt to understand the behavior of the Lisp language on shared memory machines, work was done on the QLisp system [Okuno 87b]. Although this work was not used directly by other parts of the architectures project, it investigated some of the constraints on parallelizing production systems by studying the OPS5 language.

In the search for higher performance symbolic computation, work was also done on the development of high-performance Lisp interpreters [Okuno 87a]. This work was also not used directly on the Architectures Project, since all of the code we use in our experiments has been compiled.

4. Problem-Solving Frameworks

One of the key layers in the strategy of the Architectures project was that of problem-solving frameworks. Faced with a large number of different problem-solving models the project committed itself at an early point to the Blackboard problem-solving model [Engelmore 88]. This was not an entirely arbitrary choice. The blackboard metaphor had already been applied successfully in the area of real-time signal processing [Nii 82], the selected problem domain for the AAP. It was also anticipated that the blackboard metaphor would help us to extract parallelism from the application in the way that the problems were formulated because the metaphor has a model of asynchrony built into it. For reasons detailed in [Rice 88a] the blackboard model turned out not to be as parallel as we might have hoped, but we still know of no better one for concurrent execution.

The development of problem-solving frameworks took two distinct courses. First was the development of a fairly conservative, concurrent implementation of an existing blackboard system to run on existing shared-memory machines. This was the Cage system described in Section 4.1. The second prong of the attack was to rethink that blackboard metaphor from scratch in the hope of achieving really high performance on distributed-memory multiprocessors, such as the CARE machine. This resulted in the Poligon system described in Section 4.2.

Three generations of papers have been produced describing the strategy of the Architectures project, the Cage and Poligon systems as they evolved, and the experimental results produced by these systems [Nii 86], [Nii 88a] and [Nii 88b].

4.1. Cage

Cage (Concurrent AGE) [Aiello 86] is a reimplementaion of the AGE [Nii 79] blackboard system also developed at the Heuristic Programming Project at Stanford. The central idea behind Cage is that the blackboard model provides a certain amount of parallelism by its very nature. It should therefore be possible to exploit this parallelism without any major redesign or rethink for the problem-solving model. Cage is, therefore, an implementation, which is designed to allow the concurrent execution of a blackboard system through the concurrent execution of the knowledge sources and rules in the application (see Figure 6).

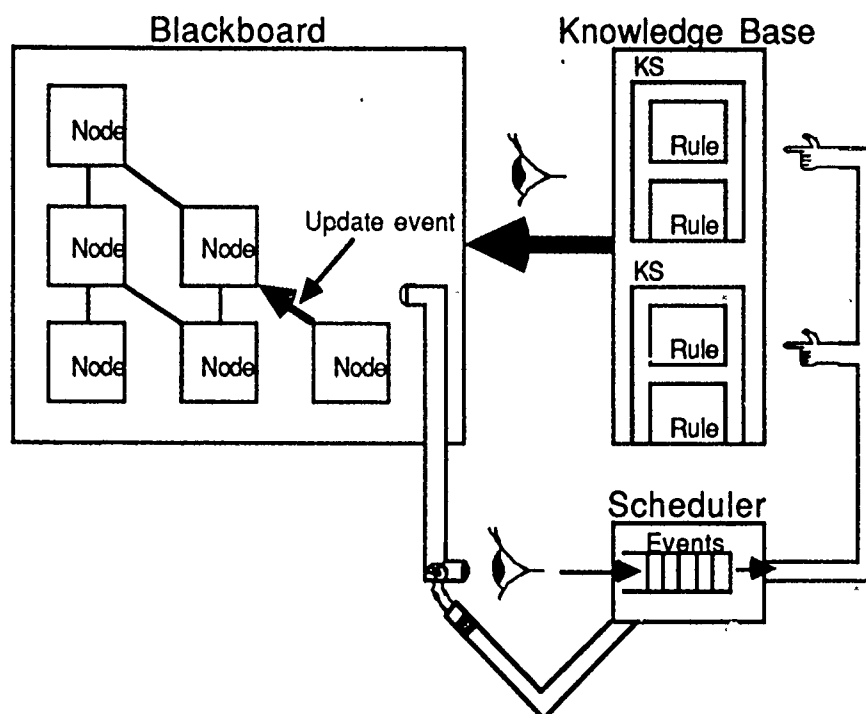


Figure 6. The Cage Architecture. Update events are perceived by the scheduling component and collected in a global event queue. The scheduler selects the knowledge sources that are interested in any given event and can execute them in parallel. These knowledge sources in turn inspect the blackboard and perform updates that are seen by the scheduler.

At the outset it was not known how difficult it would be to program such a system and how much performance could be expected, but it was thought that such an architecture might well be suitable for the current generation of multiprocessors, which mostly have a shared-memory design. Blackboard systems are typically implemented using a central, shared database to represent the blackboard. The match between the shared blackboard and the shared memory resource seemed to be worth investigating.

The Cage system was implemented first on a simple emulator, which emulated the functionality of a QLisp implementation without paying the costs of detailed simulation. It was later ported to run on the CARE simulator, using an implementation of the QLisp language built on top of the Lamina shared-variable programming interface [Saraiya 88].

The Elint application, described in Section 5.1 was mounted on the Cage system and experiments were performed on it. These are detailed in [Aiello 88] and [Rice 89a]. The Cage system has shown that blackboard programs can, indeed, be run in parallel in a relatively

simplistic manner. The performance of Cage, however, is restricted by a number of factors [Nii 88b]:

- its implementation, which was not highly tuned;
- its architecture, which exhibits significant contention for global shared resources such as the event queue;
- the QLisp substrate, on which it is built, and
- the shared-memory hardware upon which it runs.

Thus, although the Cage architecture is a viable architecture for existing shared-memory hardware systems, because of the close link between the Cage programming model and its underlying hardware, we do not anticipate that future concurrent expert system tools will be built much like Cage. We believe that the trend of multiprocessor design is broadly away from shared-memory machines and towards distributed-memory designs because of their greater ability to scale. Software design is likely to track this trend.

4.2. Polygon

The expectation that the next generation of multiprocessors, for reasons of simplicity, performance and cost, are likely to be distributed memory machines required a rethink of the blackboard model before it could be mounted on such a machine in a manner likely to deliver good performance. Polygon [Rice 86a] and [Rice 86b] was developed in an attempt to address these needs. Polygon took the view that processors were going to be cheap and plentiful and thus that if necessary it was quite acceptable to allocate one processor or more to each node on the blackboard.

First the serializing, centralized control mechanism of conventional blackboard systems was discarded. Distributing the nodes of the blackboard over the processor network allowed the knowledge base to be spread over the blackboard as well, so as to eliminate any performance bottleneck due to the communication costs between the knowledge base and the blackboard. The simplest available rule invocation mechanism was selected, so as to maximize performance; rules were directly attached to slots of the nodes on the blackboard. A modification to a slot, to which a rule was attached, resulted in that rule being invoked. Rule invocations were spun off into different processes on different processors for execution, thus minimizing the length of the critical sections on the processors holding blackboard nodes and allowing multiple, simultaneous rule invocations for the same modified blackboard object (see Figure 7).

In practice, these mechanisms did indeed result in good performance, but they also resulted in significant problems. Lots of uncontrolled asynchronous processes all reading and writing things in a shared database are bound to cause problems when it comes to getting a coherent or correct answer. Extra mechanisms had to be implemented, which allowed the blackboard nodes to have "goals" and the ability to evaluate their own performance with respect to the overall goal of the system. This allowed the blackboard nodes to have the final decision about whether to perform any modification operation attempted by a rule. The result was a sort of distributed hill-climbing behavior. Nodes iterated towards a good solution.

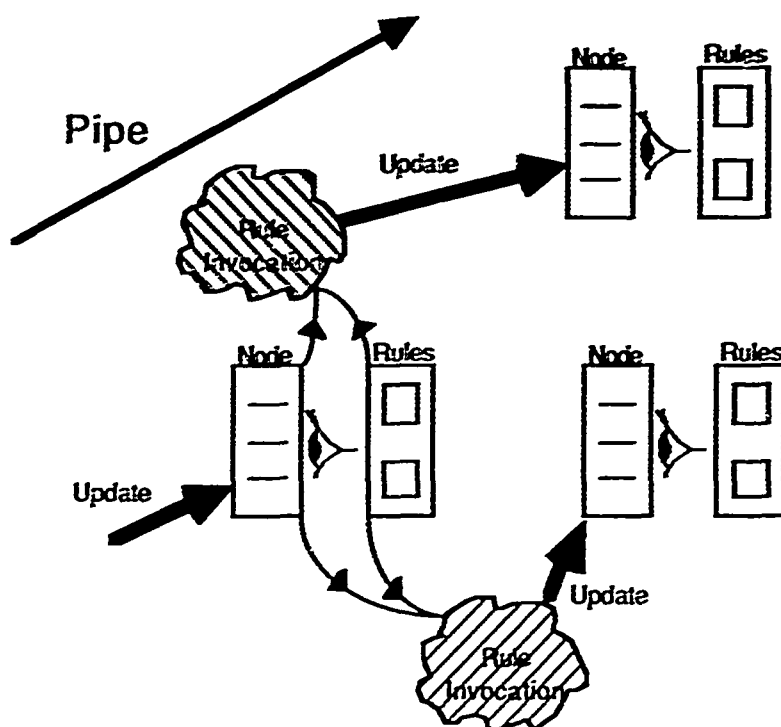


Figure 7. The Polygon Architecture. Updates on the blackboard are observed by rules which watch specific slots of blackboard nodes. These rules can fire in parallel causing further updates to the same or other nodes. This flow of updates from one node to another implicitly forms pipes, which increase the parallelism realizable by the system.

These mechanisms did not come without associated costs in terms of granularity. Although the Polygon system delivers very high performance when compared to other blackboard systems such as AGE, it nevertheless significantly lacks the performance provided by an application written directly in Lamina. Polygon, therefore, provides a fairly general concurrent implementation of the blackboard problem-solving model with all of the advantages of abstraction and modularity that this confers. It does so, however at a price. A detailed description of Polygon's design and implementation can be found in [Rice 89b], which also describes a number of means by which the performance of Polygon could be improved by superior compilation if it were to be turned into a production quality system.

The Elint application, described in Section 5.1, was implemented in the Cage, Polygon and Lamina systems. The results of these experiments are reported in [Rice 88b], [Rice 89a] and [Nii 88b]. Another application called ParAble, implemented using the Polygon framework, is described in Section 5.3.

5. Applications

As was mentioned in the introduction we expected at the outset that the project would be application driven. In the search for an application domain, which would need significant speed-up in order for expert systems to be fielded, and yet held a certain obvious potential for concurrent execution, we picked on the field of real-time signal understanding. Existing blackboard systems, such as HASP/SIAP [Nii 82] and Tricero [Williams 84] had shown both that the blackboard problem-solving model was appropriate for this domain and that the performance deliverable using the existing blackboard tools was entirely inadequate to field such systems.

What we needed, therefore, was a problem which was complex enough to give us a reasonable model of a real system, and yet simple enough that we would not spend too much effort on the mechanics of its implementation. Contrary to our original intentions of being application driven, the unavailability of a satisfactory application at the start of the project caused the project to become somewhat more hardware driven than expected. It was eventually decided that we would focus on a problem called Elint, a system for the understanding of passive radar signals. This application is described in Section 5.1

After a fair bit of experimentation it was determined that our ability to exploit parallelism was being constrained by the problem we were using - it was not sufficiently complex. In the search for a more knowledge-rich and computationally intensive application we developed the AirTrac application. This is described in Section 5.2.

Work has also been done in areas other than that of real-time signal understanding; ParAble, a system for fault-finding in particle accelerator beam lines has been developed using the Poligon framework. This is described in Section 5.3. A number of numerical or semi-numerical programs have also been developed during our more hardware-related experiments. These are mentioned in Section 5.4.

5.1. Elint

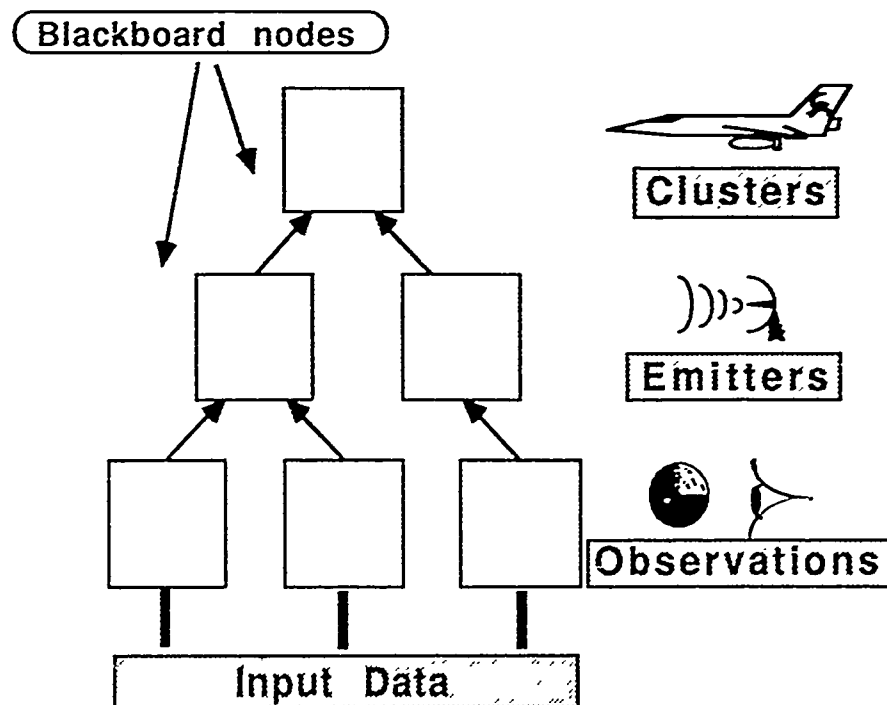


Figure 8. The Elint Application. Sensor data is abstracted into hypothetical radar emitters, which are tracked as clusters of emitters.

Elint is a soft real-time system for the interpretation of passive radar signals. Data are collected from a number of receiving stations and are integrated so as to allow the system to track radar emitting aircraft as they pass through the monitored airspace. The data are abstracted into hypothetical radar emitting platforms. These are in turn collected into clusters of emitters, which might represent a number of planes or a single plane using multiple radar systems, as is often the case with modern military aircraft (see Figure 8).

Elint was first implemented using the CAOS system. It was originally thought that this work would take only a couple of months to do. In fact, the complete task — implementation, experimentation and analysis of results — took 18 months. We learned early on that it is by no means a trivial matter to reimplement an existing, serial application in a parallel environment. These initial experiments are detailed in [Brown 86].

Since the CAOS implementation, Elint has been implemented three times; using Lamina [Delagi 88b] [Saraiya 89] and the Cage [Aiello 88] and Polygon [Rice 88b] [Nii 88b] frameworks and a number of experiments have been performed on them.

5.2. AirTrac

The development of the Elint application showed us that the amount of parallelism that could be demonstrated was much more dependent on the application than we had anticipated. We had hoped that by extracting parallelism at the different levels of the system's implementation hierarchy we could gain significant speed-up. We were unable to demonstrate this. We were able to show, however, that our experiments showed poor speed-up largely because the application itself had run out of parallelism.

What we needed was an application which would really stretch the hardware and software we were developing in a realistic manner. In response to this the AirTrac application was developed [Delaney 20].

The AirTrac problem domain sounds superficially like that of Elint. It was a system for the interpretation of radar data, though in this case the radar systems modeled were active, not passive. Unlike Elint, AirTrac was designed to go much further than simply tracking aircraft and finding likely threats. The scenario for AirTrac was the detection of "smugglers" flying across a border. The problem faced by existing radar users is that a large number of legitimate aircraft travel in the same airspace as smugglers. Smugglers may take advantage of variations in terrain in order to find areas of poor or no radar reception. They also resort to other evasive tactics.

The system was designed in a number of layers so that different implementation efforts could be decoupled. The first subsystem implemented was called the Data Association component [Nakano 87b], and is the subsystem, which most closely matches the Elint application. It was initially intended that this component would be implemented using the Polygon framework. It was found, however, that the simulation of the Polygon system for a problem as complex as AirTrac would take prohibitively long. Consequently AirTrac was implemented directly in Lamina. Substantial speed-up was shown, which seemed to increase linearly with the number of processors. This was a very encouraging result.

The second component of AirTrac, Path Association, [Noble 88a] was significantly more knowledge intensive than the first. This subsystem was also implemented directly in Lamina initially. However, programming in the raw Lamina framework was too complex and time-consuming, so a layer was built on top of Lamina, called ELMA [Noble 88b], which provided the abstraction model needed for the implementation.

The final, most abstract, component of AirTrac has not yet been implemented. We have not yet extracted all that we can learn from the second layer and we were not able to show all of the speed-up that we thought was possible in the second layer, so work is continuing in this area.

5.3. ParAble

The ParAble project [Bandini 89] was an attempt, by choosing a completely different application domain, to test the generality of the problem-solving model offered by Polygon. To do this we decided to make a parallel implementation of the ABLE system [Selig 87], developed also at Stanford.

The objective of the ABLE project was to find a fast way to diagnose particle accelerator beamlines. These large and complex machines are very prone to beam alignment problems due either to misalignment of the magnets, which steer and focus the beam, or to problems with the power supplies to those magnets, which result in the magnets not having the desired strength. These systems are so complex that it can take many months of knob-twiddling simply to commission them.

By the use of an analytic model of the transfer functions of the beam-line components, and a number of heuristics that use successive runs of the model, comparing the results with the real data to locate the faults, the ABLE system was able to find faults in such systems in about ten minutes. As particle accelerators become more complex there may well be a need to control them in real time, so although there is no immediate need for higher performance in the ABLE system, it is not unreasonable to suppose that there might be in the future.

A number of Experiments have been performed on ParAble, detailed in [Bandini 89]. The realizable parallelism in this project was, again, found to be limited mostly by the availability of data parallelism.

5.4. Numerical and Semi-numerical programs

The expert systems mentioned above are not ideal applications for multiprocessor execution. They are irregular and very data dependent. A large body of applications already exists in the area of numerical and semi-numerical processing, which will require the speed-up associated with parallel execution. Indeed, such programs are already being run on a number of multiprocessors. It is therefore essential that any machine designed with a view to being general-purpose must also be able to execute these regular, algorithmic problems efficiently. A number of small numerical programs have been developed, therefore, which allow us to test our hardware and software ideas in a much more controllable way than we can with any expert system application. Among these are a Gaussian elimination algorithm, a partial differential equation solver and an integrated circuit line simulator.

6. Conclusions

The Advanced Architectures Project has run for a number of years now. We on the project have found that we have moved from a position of having a good understanding of AI problem-solving techniques and knowing little about parallel computation to one where we now believe that we have a good understanding of the issues involved; the problems facing the implementors of concurrent problem-solving systems and the gains that they can reasonably expect. We hope that our successes and failures reported in the publications mentioned here will help the rest of the research community as we start to move over to concurrent computational platforms.

7. Bibliography of Advanced Architectures Project Publications

- [Aiello 86] Nelleke Aiello. *User-Directed Control of Parallelism; The CAGE System*. Technical Report KSL-86-31, Heuristic Programming Project, Computer Science Department, Stanford University, 1986. Also in Proceedings of DARPA Expert Systems Workshop, April 1986, Science Applications International Corp., Arlington, VA. Report SAIC-86/1701, and [Stanford 88].
- [Aiello 88] Nelleke Aiello. *Cage: The Performance of a Concurrent Blackboard Environment*. Technical Report KSL-88-80, Heuristic Programming Project, Computer Science Department, Stanford University, 1988.
- [Bandini 89] Jean-Christophe Bandini. *Poligon Applications*. Technical Report KSL-89-43, Heuristic Programming Project, Computer Science Department, Stanford University, 1989.
- [Brown 86] Harold Brown, Eric Schoen, Bruce A. Delagi. *An Experiment in Knowledge-Base Signal Understanding Using Parallel Architectures*. Technical Report STAN-CS-86-1136, Heuristic Programming Project, Computer Science Department, Stanford University, 1986. Also in [Stanford 88].
- [Byrd 87a] Gregory T. Byrd, Russel T. Nakano and Bruce A. Delagi. *A Point-to-Point Multicast Communications Protocol*. Technical Report KSL-87-02, Heuristic Programming Project, Computer Science Department, Stanford University, 1987.
- [Byrd 87b] Gregory T. Byrd, Bruce A. Delagi. *Considerations for Multiprocessor Topologies*. Technical Report KSL-87-07, Heuristic Programming Project, Computer Science Department, Stanford University, 1987. Also in Proceedings of DARPA Knowledge Based Systems Workshop, April 1987.
- [Byrd 87c] Gregory T. Byrd, Russel T. Nakano, Bruce A. Delagi. *A Dynamic, Cut-Through Communications Protocol with Multicast*. Technical Report STAN-CS-87-1178, Heuristic Programming Project, Computer Science Department, Stanford University, 1987.
- [Byrd 88a] Gregory T. Byrd, Bruce A. Delagi. *A Performance Comparison of Shared Variables vs. Message Passing*. Technical Report KSL-88-10, Heuristic Programming Project, Computer Science Department, Stanford University, 1988. Also in Proceedings of Third International Conference on Supercomputing, pages 1-7, Boston, MA, March 1988 International Supercomputing Institute.
- [Byrd 88b] Gregory T. Byrd, Nakul P. Saraiya and Bruce A. Delagi. *Multicast Communication in Multiprocessor Systems*. Technical Report KSL-88-81, Heuristic Programming Project, Computer Science Department, Stanford University, 1988.

- [Byrd 89] Gregory T. Byrd. *Support for Fine-Grained Message Passing in Shared Memory Multiprocessors*. Technical Report KSL-89-15, Heuristic Programming Project, Computer Science Department, Stanford University, 1989. Also to appear in Proceedings of the 5th Annual Computer Science Symposium, University of South Carolina, April 7-8, 1989
- [Davies 86] Davies, Byron. *CAREL: A Visible Distributed Lisp*. Technical Report KSL-86-14, Heuristic Programming Project, Computer Science Department, Stanford University, 1986. Also in Proceedings of DARPA Expert Systems Workshop, April 1986, Science Applications International Corp., Arlington, VA. Report SAIC-86/1701.
- [Delagi 86a] Bruce A. Delagi, Nakul P. Saraiya, Gregory T. Byrd. *LAMINA: CARE Applications Interface*. Technical Report KSL-86-67, Heuristic Programming Project, Computer Science Department, Stanford University, 1986. Also in Proceedings of Third International Conference on Supercomputing, pages 12-21, Boston, MA, March 1988 International Supercomputing Institute, and [Stanford 88].
- [Delagi 86b] Bruce A. Delagi, Nakul P. Saraiya, Sayuri Nishimura, Gregory T. Byrd. *An Instrumented Architectural Simulation System*. Technical Report KSL-86-36, Heuristic Programming Project, Computer Science Department, Stanford University, 1986. Also in [Stanford 88] and *Artificial Intelligence and Simulation: The diversity of Application*. The Society for Computer Simulation International, February 1988.
- [Delagi 87] Bruce A. Delagi, Nakul P. Saraiya, Sayuri Nishimura, Gregory T. Byrd. *Instrumented Architectural Simulation*. Technical Report STAN-CS-87-1189, Heuristic Programming Project, Computer Science Department, Stanford University, 1987. Also in Proceedings of Third International Conference on Supercomputing, pages 8-11, Boston, MA, March 1988 International Supercomputing Institute.
- [Delagi 88a] Bruce A. Delagi, Nakul P. Saraiya, Gregory T. Byrd and Sayuri Nishimura. *CARE User's Manual*. Technical Report KSL-88-53, Heuristic Programming Project, Computer Science Department, Stanford University, 1988.
- [Delagi 88b] Bruce A. Delagi and Nakul P. Saraiya. *ELINT in LAMINA: Application of a Concurrent Object Language*. Technical Report KSL-88-33, Heuristic Programming Project, Computer Science Department, Stanford University, 1988. Also in SIGPLAN Notices, February 1989.
- [Delaney 86] John R. Delaney. *Multi-System Report Integration Using Blackboards*. Technical Report KSL-86-20, Heuristic Programming Project, Computer Science Department, Stanford University, 1986. Also in Proceedings of DARPA Expert Systems

Workshop, April 1986, Science Applications International Corp., Arlington, VA. Report SAIC-86/1701, and [Stanford 88].

- [Hailperin 88] Max Hailperin. *Load Balancing for Massively Parallel Soft-Real-Time Systems*. Technical Report KSL-88-62, Heuristic Programming Project, Computer Science Department, Stanford University, 1988. A condensed version of appears in the proceedings of "Frontiers '88: The Second Symposium on the Frontiers of Massively Parallel Computation."
- [Nakano 87b] Russel T. Nakano, Masafumi Minami. *Experiments with a Knowledge-Based System on a Multiprocessor*. Technical Report KSL-87-61, Heuristic Programming Project, Computer Science Department, Stanford University, 1987. Also in a shortened form in Proceedings of Third International Conference on Supercomputing, pages 22-24, Boston, MA, March 1988 International Supercomputing Institute, and [Stanford 88].
- [Nii 86] H. Penny Nii. *CAGE and POLIGON: Two Frameworks for Blackboard-based Concurrent Problem Solving*. Technical Report KSL-86-41, Knowledge Systems Laboratory, Computer Science Department, Stanford University, 1986. Also in Proceedings of DARPA Expert Systems Workshop, April 1986, Science Applications International Corp., Arlington, VA. Report SAIC-86/1701.
- [Nii 88a] H. Penny Nii, Nelleke Aiello, James Rice. *CAGE and Poligon: Two Frameworks for Concurrent Problem Solving*. Technical Report KSL-88-02, Heuristic Programming Project, Computer Science Department, Stanford University, 1988. Also in [Engelmore 88], and [Stanford 88], and proceedings of AAAI 88 Blackboard Workshop.
- [Nii 88b] H. Penny Nii, Nelleke Aiello, James Rice. *Experiments on Cage and Poligon: Measuring the performance of Parallel Blackboard Systems*. Technical Report KSL-88-66, Heuristic Programming Project, Computer Science Department, Stanford University, 1988. Also in *Distributed Artificial Intelligence II*. L. Gasser and M. N. Huhns (eds). Pitman Publishing Ltd. and Morgan Kaufmann, 1989.
- [Noble 88a] Alan C. Noble and Evertt C. Rogers. *AIRTRAC Path Association: Development of a Knowledge-Based System for a Multiprocessor*. Technical Report KSL-88-41, Heuristic Programming Project, Computer Science Department, Stanford University, 1988.
- [Noble 88b] Alan C. Noble. *ELMA Programmer's Guide*. Technical Report KSL-88-41, Heuristic Programming Project, Computer Science Department, Stanford University, 1988.
- [Okuno 87a] Hiroshi Okuno, Nobuyasu Osato and Ikuo Takeuchi. *Firmware Approach to Fast Lisp Interpreter*. Technical Report STAN-CS-87-1184, Heuristic Programming Project, Computer Science Department, Stanford University, 1987.

- [Okuno 87b] Hiroshi G. Okuno, Anoop Gupta. *Parallel Execution of OPS5 in QLISP*. Technical Report KSL-87-43, Heuristic Programming Project, Computer Science Department, Stanford University, 1987.
- [Rice 86a] James Rice. *Poligon, A System for Parallel Problem Solving*. Technical Report KSL-86-19, Heuristic Programming Project, Computer Science Department, Stanford University, 1986. Also in Proceedings of DARPA Expert Systems Workshop, April 1986, Science Applications International Corp., Arlington, VA. Report SAIC-86/1701, and [Stanford 88].
- [Rice 86b] James Rice. *The Poligon User's Manual*. Technical Report KSL-86-10, Heuristic Programming Project, Computer Science Department, Stanford University, 1986.
- [Rice 88a] James Rice. *Problems with Problem-Solving in Parallel: The Poligon System*. Technical Report KSL-88-04, Heuristic Programming Project, Computer Science Department, Stanford University, 1988. Also in Proceedings of Third International Conference on Supercomputing, pages 25-34, Boston, MA, March 1988 International Supercomputing Institute, *Artificial Intelligence, Simulation and Modelling*, Lawrence Widman (ed), John Wiley Publishing Company, New York 1989, and [Stanford 88].
- [Rice 88b] James Rice. *The Elint Application on Poligon: The Architecture and Performance of a Concurrent Blackboard System*. Technical Report KSL-88-69, Heuristic Programming Project, Computer Science Department, Stanford University, 1988. Also in Proceedings of IJCAI 89.
- [Rice 88c] James Rice. *The Advanced Architectures Project*. Technical Report KSL-88-71, Heuristic Programming Project, Computer Science Department, Stanford University, 1988.
- [Rice 89a] James Rice and Nelleke Aiello. *See How They Run... The Architecture and Performance of Two Concurrent Blackboard Systems*. Technical Report KSL-89-08, Heuristic Programming Project, Computer Science Department, Stanford University, 1989. Also in *Blackboard Architectures and Applications: Current Trends*, V. Jagannathan and R. Dodhiawala (eds), Academic Press, 1989.
- [Rice 89b] James Rice. *The Design of a High Performance, Concurrent Problem Solving System...and many Lessons Learned on the Way*. Technical Report KSL-89-37, Heuristic Programming Project, Computer Science Department, Stanford University, 1989.
- [Saraiya 86] Nakul P. Saraiya. *AIDE: A Distributed Environment for Design and Simulation*. Technical Report KSL-86-56, Heuristic Programming Project, Computer Science Department, Stanford University, 1986. Also in Proceedings of DARPA Expert Systems Workshop, April 1986, Science Applications International Corp., Arlington, VA. Report SAIC-86/1701 (preliminary version).

- [Saraiya 88] Nakul P. Saraiya. *A Shared Memory Lisp Package for CARE*. Technical Report KSL-88-85, Heuristic Programming Project, Computer Science Department, Stanford University, 1988.
- [Saraiya 89] Nakul P. Saraiya. *Design and Performance Evaluation of a Parallel Report Integration System*. Technical Report KSL-89-16, Heuristic Programming Project, Computer Science Department, Stanford University, April 1989.
- [Schoen 86] Eric Schoen. *The CAOS System*. Technical Report KSL-86-22, Heuristic Programming Project, Computer Science Department, Stanford University, 1986. Also in Proceedings of DARPA Expert Systems Workshop, April 1986, Science Applications International Corp., Arlington, VA. Report SAIC-86/1701.
- [Stanford 88] Expert Systems on Multiprocessor Architectures: Phase One Final Report. Rome Air Development Center RADC-TR-88-187.

8. Bibliography of Referenced Work Not Performed on the Advanced Architectures Project

- [Abelson 83] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA 1983.
- [Engelmore 88] Robert Engelmore and Tony Morgan (eds.) *Blackboard Systems*. Addison-Wesley Publishing Company Inc., Menlo Park 1988.
- [Gabriel 84] Richard P. Gabriel and John McCarthy. *Queue-based Multi-processing Lisp*. Proceedings of the ACM Symposium on Lisp and Functional Programming: 25-44, August, 1984.
- [Gupta 86] Anoop Gupta. *Parallelism in Productions Systems*. Technical Report, Computer Science Department, Carnegie-Mellon University, March, 1986. Ph. D. dissertation.
- [Nii 79] H. Penny Nii and Nelleke Aiello. *AGE: A Knowledge-based Program for Building Knowledge-based Programs*. Technical Report HPP-79-4, Heuristic Programming Project, Computer Science Department, Stanford University, 1979. Also in Proceedings of the 6th International Joint Conference on Artificial Intelligence: 645-655, 1979.
- [Nii 82] H. P. Nii, E. A. Feigenbaum, J. J. Anton, and A. J. Rockmore. *Signal-to-Symbol Transformation: HASP/ISIAP Case Study*. Technical Report HPP-82-6, Heuristic Programming Project, Computer Science Department, Stanford University, 1982. Also in AI Magazine. 3:2, 23-35, 1982.
- [Selig 87] Lawrence J. Selig. *An Expert System using Numerical Simulation and Optimization to find Particle Beam Line Errors*. Technical Report KSL-87-36, Heuristic Programming Project, Computer Science Department, Stanford University, 1987.

[Williams 84]

Mark Williams, Harold Brown and Terry Barnes. *TRICERO
Design Description*. ESL Inc. 1984.

See How They Run...
The Architecture and Performance of Two
Concurrent Blackboard Systems

by
James Rice and Nelleke Aiello
(Rice@Sumex-Aim.Stanford.Edu and Aiello@Sumex-Aim.Stanford.Edu)

Knowledge Systems Laboratory
Stanford University
701 Welch Road, Bldg. C,
Palo Alto, CA 94304

The authors gratefully acknowledge the support of the following funding agencies for this project; DARPA/RADC, under contract F30602-85-C-0012; NASA, under contract number NCC 2-220; Boeing Computer Services, under contract number W-266875.

Abstract

In this paper we discuss Poligon and Cage, two skeletal systems for the development of concurrent blackboard based applications, their architecture and the motivation for their design. A number of experiments have been performed in order to evaluate the performance of these systems. Some of these are detailed and the results derived are given. Lessons learned in the development of both Cage and Poligon are given and conclusions about the performance of similar systems are drawn.

1. Introduction

It is often said that future AI applications will make significantly greater computational demands than the present generation. The Advanced Architectures Project of Stanford University's Heuristic Programming Project [Rice 88b] is investigating this issue, since it has as its objective achieving computational speed-up for expert systems through the use of parallel hardware and new, advanced software architectures. This requires the development of everything from designs for parallel hardware, which might be appropriate for the execution of future symbolic programs, through operating system and language concepts to problem-solving frameworks and eventually mounting applications on them in order to test the new designs.

Poligon [Rice 86] and Cage [Aiello 86] are two problem-solving frameworks developed as part of the Advanced Architectures Project. Cage uses parallelism at the problem solving level and is further constrained to a target system architecture of shared-memory multiprocessors. Poligon is designed to exploit distributed-memory machines and has a granularity tuned to that of its underlying hardware. The potential applications envisioned for this work can be characterized as performing real-time interpretation of continuous streams of errorful data, a class of applications that currently run too slowly on serial blackboard systems to be of practical use. In Sections 2 and 3 we describe the Cage and Poligon systems and their architectures as designs for high-performance, concurrent blackboard¹ systems aimed particularly at the problem domain of soft real-time problems and the motivation for these designs. Section 4 discusses the applications mounted on the Poligon and Cage frameworks and experiments performed on these systems to measure their performance, showing the results of these experiments in Sections 4.3.1 and 4.4.1 and briefly interpreting these results. We conclude in Section 5 with a number of the lessons we have learned in the process and pointers for future research.

The field of parallel computing, like that of AI has its own set of buzz-words. Of particular importance to the understanding of this paper are the following:

- *Amdahl's Limit*: This is the limit to the amount of available parallelism in a program. This limit tends to be surprisingly low for most programs. If, for instance, a program has only 1% of its code that *must* be run serially, perhaps due to data dependencies, then even with an infinite number of processors, the program can only be sped up by a factor of a hundred.
- *Distributed-memory*: This is a class of multiprocessors that has a collection of processor-memory pairs, communicating with one another over some network. Typically the communication happens through message-passing, and the processing elements can of-

¹It is assumed that the reader is familiar with the blackboard model and the relevant terminology. For more information, the reader is referred to [Engelmore 88].

ten only see their neighbors in the network. There is no concept of a global shared resource.

- **Parallelism:** This is the degree to which more than one thing happens at the same time. Parallelism can manifest itself both as *pipe-line* parallelism, in which a series of different operations are applied to a sequence of data, like a car assembly line, and *replication*, in which largely similar operations are performed on separate pieces of data, like a number of men, each building a complete engine for a car.
- **Serialization:** This is the state of one computational activity following another in time. This is what happens when processes become synchronized. All programs have a serial component. Our task is to minimize this.
- **Shared-memory:** This is a class of multiprocessors that has a set of processors, often with local caches, which all have equal access to a shared memory resource. Simple implementations of this model connect a number of processors and memory controllers to a bus.
- **Synchronization:** This term is used to describe that event which brings asynchronous, parallel processes together synchronously. Synchronization primitives are used to enforce serialization. For instance, if both of the operands for a "+" operation are being computed in parallel then there must be a point of synchronization at the point when the addition actually takes place. The process doing the addition must *wait* until both of the operands have arrived.

2. The Cage System

The Cage system is an extension of the serial AGE system [Nii 79]. The two systems are identical except that Cage allows parallel execution of many of its applications' components. Parallel execution in Cage can occur at different levels of granularity, based on natural divisions in the blackboard model. In this section, we will first give some background information about AGE, and then we will describe Cage and how the user can specify concurrency in Cage.

2.1. The Derivation of Cage from AGE

AGE is an implementation of a serial blackboard system framework. It is composed of a knowledge base, in the form of *knowledge sources* (KSs), and a structured solution space, a *blackboard*, where the KSs can post interim results and read the results of other KS executions. KSs contain *condition-action rules* that can read the blackboard and make changes on it. The blackboard is a structured set of *levels* on which objects are created and modified by the rules. These changes to the blackboard are called *events*. A scheduling mechanism, or *controller*, programmable by the user, invokes one KS at a time from among those *triggered* by the preceding events. Figure 1 shows the control and data flow of this serial control cycle.

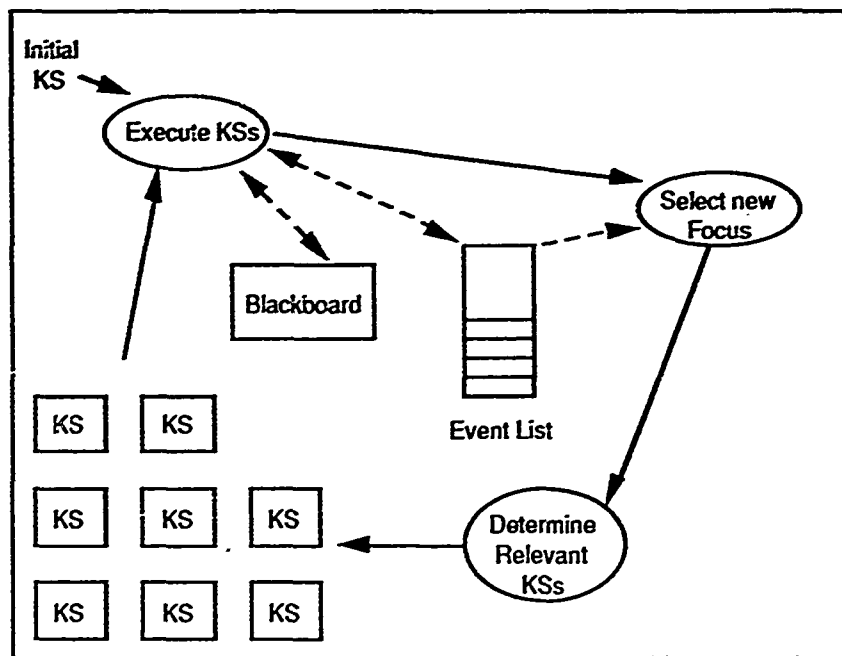


Figure 1. AGE Serial Control Cycle.

2.2. Cage Architecture

The basic components of Cage are the same as AGE's with one addition — the declarations that specify which components to execute in parallel and at which points to synchronize. The components which can be executed in parallel in Cage are the KSs, the rules within the KSs, and the condition and action parts of rules. Synchronization points can be specified (1) in the control cycle between the concurrent execution of KSs, (2) within a KS after evaluating all the rules' conditions but before executing any actions, or (3) within a rule, between the condition and action parts. By selecting one of the concurrency control options, the user can alter the simple, serial execution of KSs and their components so that they are executed in parallel. Next we will discuss each potential source of concurrency in more detail.

2.2.1. Knowledge Source Concurrency

Two possible sources of concurrency exist at the KS level. A number of KSs can work either on different parts of the blackboard at the same time or in a pipeline fashion. In the application area of real-time interpretation of data, many instances of the same KS can simultaneously deal with new data items. Each of these KSs then becomes the first in a chain of KSs which interprets the data up the blackboard's levels of abstraction.

KSs in Cage can be executed in parallel with or without synchronization at the control level. With synchronization, the controller waits for all previously invoked KSs to complete before invoking the next set of triggered KSs. Without synchronization, KSs are invoked immediately when triggered, without waiting for any other KS.

2.2.2. Rule Concurrency

Within each KS further concurrency is possible by executing the rules in parallel. Again, Cage provides several different options for running the rules in parallel. First the condition parts of rules are evaluated. Next, if the user opts to synchronize, the controller will wait until all the conditions have been evaluated before executing the action parts of the applica-

ble rules concurrently. The user can also specify the parallel evaluation of the conditions with the serial execution of the actions. Without synchronization, the applicable actions are executed as soon as a rule's conditions have been evaluated.

2.2.3. Clause Concurrency

Even finer grain concurrency is possible in Cage within each rule, by executing individual predicates of the condition part concurrently. Only one option is available; evaluation of the predicates in parallel and execution of the action clauses in the action part of applicable rules in parallel.

2.3. Using Cage

In addition to the speed-up and throughput data about Cage gathered in the experiments described in the Section 4, we also learned a number of lessons about programming in a concurrent environment. Implementing the concurrency outlined above created a number of programming problems. For example, at the rule level, the state of the blackboard which leads to a rule firing may be changed before that rule's actions can be executed (data inconsistency). Also, a rule may access values from several different blackboard objects with no guarantee that those values are consistent with each other (data incoherence). Memory contention can be a problem at the clause level, if a number of clauses refer to the same blackboard object at the same time, negating the benefits of concurrent execution.

Data inconsistency was alleviated by creating an atomic operation that could read from and then write to a blackboard object without allowing any intervening operations. In addition, a block-read (read several slots in a block) operation was defined, so that a rule can read all relevant information from an object with the guarantee that data will be consistent within that object. No other operations are allowed to an object during a block-read of that object.

Data coherence can be maintained when running KSs in parallel, by reading all the slots of an object that are referenced in a KS at the same time, locking the object just once. This is in contrast to locking the object every time a slot is read by the rules. In other words, all necessary blackboard data is collected into local variables, called *definitions* in the KS's activation context before any rules are executed. Thus all the rules within a KS refer to data viewed at the same time.

In a serial blackboard system one KS precondition may serve to describe several changes to the blackboard adequately. For example, suppose the firing of one rule causes three changes to be made serially. The last change, or event, is usually a sufficient precondition for the selection of the next KS. In a concurrent system, however, since those changes may occur asynchronously, all three events must be included in a KS's precondition to ensure that all three changes have actually occurred before the KS is executed. In general, a simple precondition consisting of an event token is not sufficient as it was in a serial system. A detailed specification of the activation requirements of the KSs must be available, either in their preconditions or in the controller.

Occasionally two KSs running in parallel may attempt to change a slot at almost the same time. It is possible that the first change could invalidate the later changes. To overcome this *race condition*, a conditional action — an action which checks the value of a slot before making a change — was added.

3. The Polygon Architecture

In this section we briefly discuss the architecture of the Polygon system. A more detailed description of the design rational for Polygon can be found in [Nii 88].

When we started the Advanced Architectures Project we suspected that the blackboard problem-solving architecture might offer a basis for the efficient exploitation of concurrent hardware. This was because the blackboard model appeared to have concurrency built into it. Why this is, in fact, not the case is explained in [Rice 88a]. The primary reasons why the blackboard model of a collection of simultaneously cooperating experts cannot develop the parallelism that one might expect is that the blackboard model itself assumes effectively infinite bandwidth with which the experts can see any part of the blackboard that might be of interest. It also assumes that experts do not get into one another's way while solving the problem. In practice, a knowledge source can only see a small segment of the blackboard at any one time without degrading the performance of the system unacceptably. Similarly, the experts are dependent on one another; they must often wait for the results deduced by other agents and can be confused by updates being posted at unexpected times or in surprising orders. We are, however, unaware of a better architecture for concurrent problem-solving than that of Blackboard systems.

Although a number of other research efforts have looked at concurrent blackboard systems, these have concentrated primarily on either the aspects of distributed, concurrent problem-solving, such as [Lesser 83] or on coarse grained parallel systems, such as [Fennell 77] or [Ensor 85]. Polygon is a finer grained system than these, directed particularly at gaining speed-up through parallel execution.

The normal, serial implementations of the blackboard metaphor use a scheduling mechanism to cause one rule to fire after another. In parallel systems it is crucial that the programmer eliminate serial components, since this limits speed-up.¹ The main motivation of the Polygon system was to find a way to eliminate the serializing aspects of the blackboard model. We viewed this as doing the following:

- Eliminating the scheduling mechanism and finding ways to support concurrent rule activation all across the blackboard.
- Optimizing the design for distributed-memory, message-passing hardware, which should be able to deliver the best performance for large numbers of processors (of the order of hundreds to thousands.)
- Distributing the knowledge base over the blackboard so that there would be no serialization in the access to the blackboard from the executing knowledge.
- Designing the system so as to allow it to be highly compilable. It was clear from the outset that a considerable portion of the expense of existing AI systems is due to the fact that they are optimized for easy modification and debugging, rather than high run-time performance. The resulting system, therefore, had to be designed so as to be able to be compiled efficiently yet still be intelligible and debuggable during the development cycle.

As these ideas progressed we developed the notion of a blackboard consisting of active nodes, tightly associated with the knowledge relevant to them.

¹Speed-up can be viewed as the ratio of the system's speed using N processors to its speed using only one.

A very simple scheme was developed for invoking the knowledge that had been distributed to the blackboard nodes: rules are activated as daemons as a result of modifications to the slots of a node (see Figure 2).

The distributed-memory hardware model, on which the Poligon system was to operate, had the property that each processor was effectively a uniprocessor system. This meant that if we viewed the blackboard with a "*Node as a Process/Processor*" model then we would lose potential parallelism due to being able to execute only one piece of code (rule) at a time for any given node.

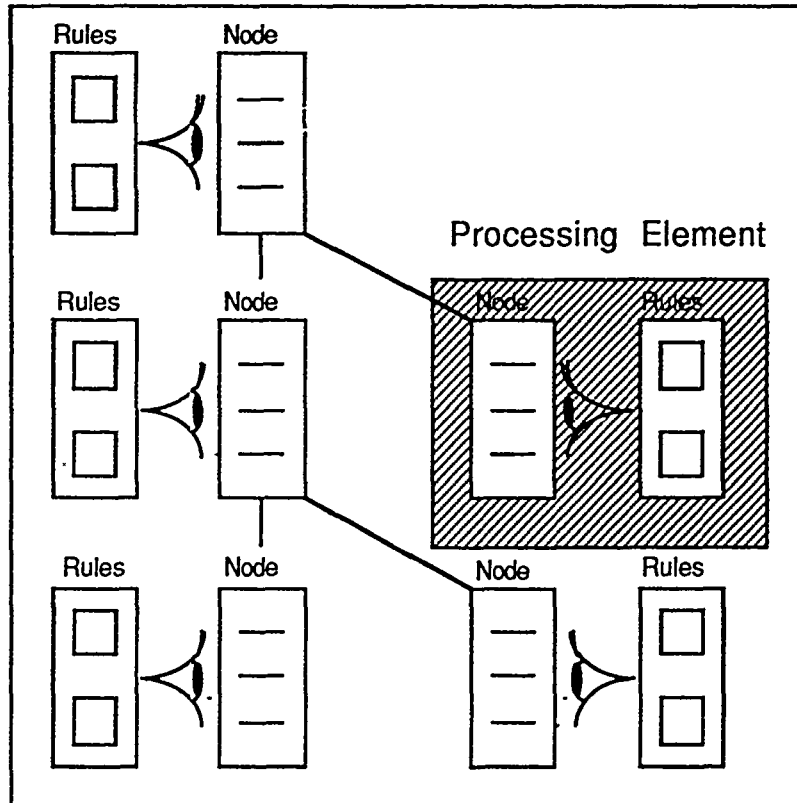


Figure 2. The Organization of the Poligon Blackboard. Rules are distributed over the network of processors and are attached to the blackboard nodes so that they can watch for modifications made to the slots in which they are interested.

What we needed, therefore, was a mechanism to allow the activation of multiple rules for any given blackboard node. This caused us to develop a model of Poligon which was as follows: A blackboard node is a process on a processor, surrounded by a collection of processors able to service its requests to execute rules. It can easily be seen that this model is very close to a distributed object system model. This is by no means a coincidence. The underlying hardware system on which Poligon was implemented was designed to support a concurrent, distributed object-oriented programming model [Delagi 86b].

The model expressed above is not without problems. In order to minimize the probability of a node being locked for a long period, which would delay remote access to it, as much processing is done in the remote rule invocations as possible.¹ This means that, when the

¹There are no user accessible locks in Poligon. Poligon nodes become locked (enter critical sections) during slot reads and updates, which are cheap operations. The Poligon architecture is such that deadlock

rules execute, they have to do so in the context of a snap-shot of the solution state as it was when the rule was invoked (see Figure 3). Remote reads to other nodes, even the invoking node, are expensive, and one cannot guarantee that things haven't changed by the time the result of the read operation has been returned.

This led to the development of the idea of a Poligon node as being an agent capable of evaluating its own performance. Mechanisms had to be included so as to allow the system to be able to assess any request to modify its local state and to decide whether to perform the update, or what else to do instead, on the basis of its own view of how it is progressing towards its goal of solving the problem.

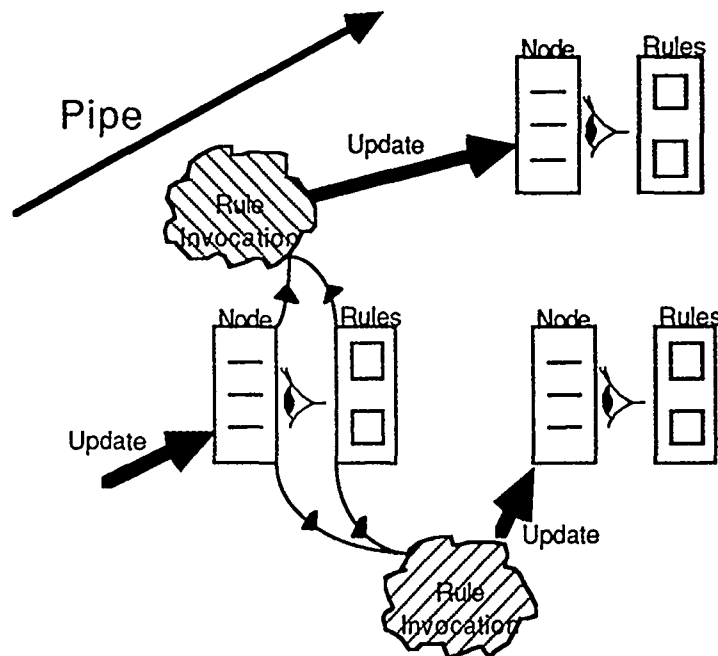


Figure 3. Updates to Poligon nodes cause concurrent rule execution, which themselves cause further updates. This implicitly forms pipes on the blackboard as data flows up or down the abstraction hierarchy.

4. Experiments

In this section we describe some of the experiments performed on the Poligon and Cage systems. Results are given for these experiments and these results are interpreted.

4.1. The Elint Application

All of the experiments reported here were performed using the same application: *Elint*, a soft real-time situation assessment problem. A more detailed treatment of the Elint experiments can be found in [Nii 88].

The Elint application encodes knowledge used to interpret the radar emissions made by planes that are received by ground-based tracking stations distributed across the country. Because these tracking sites are passive devices, they can only detect the bearing and spectral characteristics of the radar emissions. Between them, it is their responsibility to deduce

will not happen as a result of system action, though the user can still write a program that will live-lock, e.g. two nodes each waiting for one node to update the other will wait forever.

a position, course, identity, and intention for any aircraft traveling through the monitored airspace. The Elint application simulates a central machine that integrates reports from these detection sites in order to achieve the overall goals mentioned.

The important characteristics of the Elint problem were:

- A continuous stream of input data.
- No *a priori* knowledge of the behavior or number of the aircraft being tracked.
- The need to emit periodic reports capturing the system's evolving view of the solution.

The Elint problem was chosen both because it was non-trivial and was in a class of problems, for which blackboard systems had already been used, and also because it was hoped that parallelism would be readily available. It was anticipated that parallelism could be extracted from the concurrent execution of knowledge on any given part of the solution space and from the potentially large number of independent elements in that solution space, i.e. aircraft.

The application was taken from a serial implementation and was not restructured so as to be better suited for parallel execution in either the Cage or the Poligon implementation. The blackboard was composed of three distinct layers in the abstraction hierarchy. Data flowing from one level to the next allowed pipes to be formed that were three stages long.

4.2. Experimental Method

Perhaps the most important lesson that we learned from performing these experiments was to find a way to measure the relative performance of concurrent real-time systems. The methods used in the experiments changed over time, based on the results of earlier experiments. In the first experiment speed-up was measured very simply, dividing the time for the application to run a given set of input data on one processor by the time for the same system executed on multiple processors. This speed-up measure did not work well, however, because the behavior of the system changed depending on how heavily or lightly it was loaded. A rate of data arrival which adequately loaded a 4 processor machine caused data starvation for 16 processors. Later experiments used a more fair comparison scheme in which different sampling intervals were used for different numbers of processors. The sampling interval for a particular number of processors was set to be the shortest interval which still produced non-increasing latencies, where latency is the time between the input of data and the output of reports based on that data. Speed-up was measured by comparing these sampling intervals with the uniprocessor sampling interval. The sampling intervals are indicators of the throughput for a particular number of processors.

All measurements generated by the experiments were provided by the underlying CARE simulator [Delagi 86a]. Because CARE primarily simulates distributed memory architectures, it was necessary to emulate the shared memory model for the experiments on Cage by using half the processing elements (processor-memory pairs) for processors and the other half as memory controllers.¹ A variation of Qlisp [Gabriel 84], a queue based Lisp including *QLets* (parallel Let clause evaluation) and *QLambdas* (process closures), was cre-

¹In the experimental description, the "number of processors" refers to the number of processors used for processing and does not include those used for memory only.

ated to program the concurrency for Cage, whereas Poligon used a form of distributed object system supported by the CARE architecture.¹

4.3. Experiments on Cage

In this section we describe seven experiments conducted with Cage,² and the results derived from these experiments. The purpose of the experiments was to determine the speed-up and throughput achievable by Cage under various conditions, concurrency specifications, and resource allocation schemes. The first four experiments measured the speed-up gained by executing various blackboard components in parallel. The last three experiments related to improving the throughput of the Cage system.

Two different input data sets for the Elint application were used in the experiments described here. The first, called *Lumpy*, was a realistic data set with data inconsistencies, errors, and a varying number of observations per time interval. The problem with this data set was the variation in data density that made it very difficult to measure performance. A second data set, *Fat*, with a constant data density was created to deal with these problems.

4.3.1. Experiments on Speed-up and Throughput

In this section we show the experimental results from the experiments on the Cage system running the Elint application.

4.3.1.1. Experiment 1

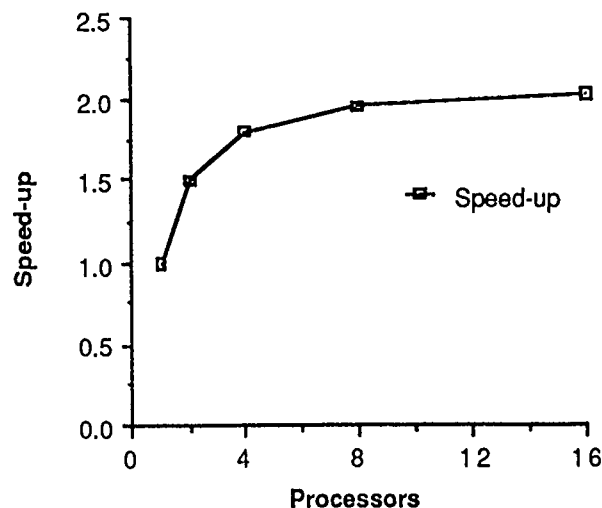


Figure 4. Results of Experiment 1.

This experiment measured the speed-up attainable for a varying numbers of processors with parallel KSs. For this experiment the controller started all triggered KS executions in parallel, waiting until they were done before selecting another set to run in parallel. Using the realistic *Lumpy* data set, this experiment exercised all the problem solving capabilities

¹The CARE shared-memory model, unlike the distributed-memory model is not a particularly sophisticated simulation. For instance, automatic data caching is not supported, though user specified caching of shared values in local memory is allowed. We believe, however, that our simulations are reasonably representative of not too sophisticated, generic shared-memory machines.

²A more complete description of these experiments can be found in [Nii 88].

of the Elint application. Experiment 1 was run serially on one processor and then over multiprocessors varying from 2 processors to 16 processors. By comparing the time required to run the data set on one processor with the time required to run with 2–16 processors, a measure of speed-up was obtained.

As shows in Figure 4, the basic speed-up began to level off with 4 processors and reached 2 with 8 processors. To explain why only a factor of two speed-up was achieved, we need to look at the serial case. In the serial case (see Figure 5) the controller selects one KS to execute from among all the KSs applicable at that time.

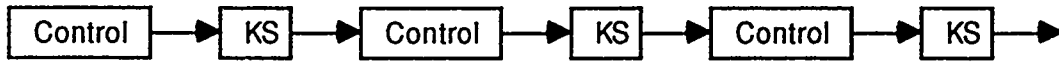


Figure 5. Basic Control Cycle for a Serial Blackboard System.

In Experiment 1 all the pending KSs are executed in parallel, as seen in Figure 6.

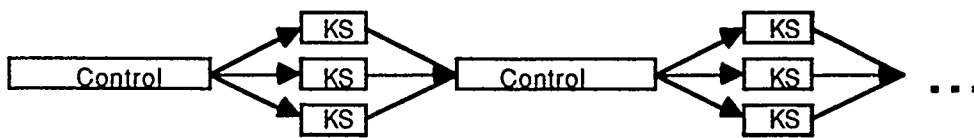


Figure 6. Basic Cycle with Serial Control and Parallel KSs in Cage.

Although the KSs were run in parallel, "Amdahl's limit" limits the speed-up to the longest serial component, in this case the controller plus the longest KS. When all component parts of the Cage execution were individually timed, it was found that in the multiprocessor case slightly less than half of the execution cycle time was being spent in the serial, synchronizing controller. Experiment 1 demonstrates that no matter how many KSs are run in parallel, speed-up gains are limited by the duration of the synchronizing controller and the KSs.

4.3.1.2. Experiment 2

Experiment 2 also measured speed-up but in a manner that was felt to be more fair than the basic speed-up experiment, using the second speed-up measure explained in Section 4.2. This and subsequent experiments used the *Fat* data set. Experiment 2 was run with three different sizes of multiprocessor, 1, 4, and 8 processors. Because of what was learned in Experiment 1, in this experiment the KSs were executed without synchronization, reducing the waiting time in the controller. As each KS completed, the controller immediately invoked any newly triggered KSs without waiting for any other KSs to finish.

The speed-up obtained by running KSs concurrently without synchronization was slightly less than 4 (see Figure 7). This is almost double the speed-up obtained with synchronization. The time spent in the controller was reduced to almost half of that in Experiment 1. But, it should be noted that the central controller is still a bottleneck. Given the architecture of blackboard systems, centralized controller time can be reduced but not eliminated without a major shift in the way we view blackboard systems.

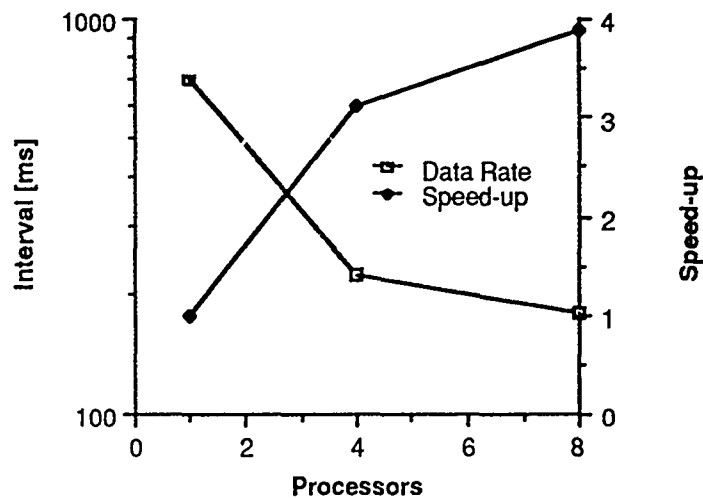


Figure 7. Results of Experiment 2.

4.3.1.3. Experiment 3

This experiment attempted to increase the speed-up by exploiting parallelism at a finer granularity than in Experiment 2. We hoped to gain a multiplicative increase in the overall speed-up for each KS by executing the rules in parallel. The rules were executed with both condition and action parts running concurrently and without synchronizing between the condition and action parts. Otherwise the experimental variables of Experiment 3 are identical to those of Experiment 2.

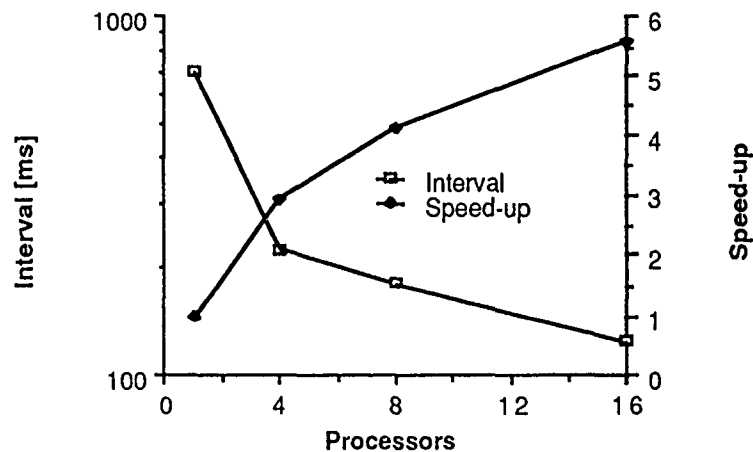


Figure 8. Results of Experiment 3.

The initial results of Experiment 3 were disappointing. For 8 processors only a 5.5% speed-up over Experiment 2 was attained, giving a total speed-up of 4.12. For 4 processors there was no speed-up at all over Experiment 2. The overhead of spawning processes offset any gains from more parallelism. We tried running Experiment 3 on a 16 processor system in the hope of alleviating the congestion on the smaller grids. This resulted in slightly better results, a total speed-up of 5.6 (see Figure 8). This extra speed-up is due to the greater availability of free processors to handle the greater number of processes produced with rule level granularity.

Throughout the first three experiments one troubling aspect was the apparent low sampling intervals Cage could support. (The sampling interval gives a measure of the actual

throughput rate.) The minimum sampling intervals for Elint on Cage were around 120ms which was considerably slower than that of other concurrent Elint applications, such as the one done on Poligon. To determine the reasons for this slow throughput, various timings on all components parts of Cage were taken. As expected, most of the time was being spent setting-up and executing KSs. However, within the KSs a very high percentage of time was spent in the creation/match process — searching for existing blackboard objects or creating new ones if no match was found. A separate creation processor handles this creation/match process in Cage. A second interesting observation was that the timings were not regular, they were, in fact, very spiky. Operations that on average took only a few milliseconds occasionally took a hundred milliseconds or more. An initial hypothesis was that the spikes were caused by blocked and descheduled processes, an indication of problems in resource allocation.

4.3.1.4. Experiment 4

Experiment 4 attempted to solve both the spikiness problem and the unexpectedly high cost of creation by allocating some of the processors to specific tasks, thus freeing those processors from interruption by other tasks. The three most time consuming tasks were creation/match, control, and data input, so these three processes were preallocated to specific processors and no other processes were allowed to run on those processors.

The results of this experiment were not conclusive. Experiment 4 had a speed-up of 3% over Experiment 3, or a total speed-up of 5.7x. But 3% falls within the margin of error for these measurements. The queue lengths for KSs and object creation/match processors increased, indicating (1) that insufficient numbers of processors were available for the KSs, because of the three preallocated processors and (2) that the object creation/match handler probably needed two or more processors to handle its load.

4.3.1.5. Experiment 5

Experiment 5, a second experiment involving specialized processor allocation, was more successful. In this case only one processor, the input-handler, was used to execute the entire input procedure. Previously the creation of new input objects (observations), one for each input data item, had been handled by a separate creation handler. By eliminating the cost of spawning the separate creation process and the possibility of blocking the input process while waiting for the creation to complete, the input object creation time was decreased by 59%. Also, the spikiness in the creation measurements almost disappeared.

One other improvement made in Experiment 5 involved the use of a new, more accurate version of the CARE simulator. Because of improvements to the design of the simulated machine, it had 4 times faster simulated memory access. This improved the total throughput by 43%. The combination of local creation by the input handler and optimizations in the simulator improved the best sampling interval in Experiment 5 from 120ms to 40ms.

4.3.1.6. Experiment 6

In Experiments 6 and 7 the number of processors used was increased to 32. Preliminary runs showed little improvement in throughput due *just* to the increased number of available processors. To use those additional processors Experiment 6 also increased the number of creation process handlers from 1 to 4. Separate processors were used to handle the creation of objects at different levels of the blackboard. These multiple creation-handler processors, together with the 16 additional processors, reduced the throughput to 31ms, a 22% improvement over the best results of Experiment 5. This improvement is a strong indication that the single creation process was a bottleneck.

4.3.1.7. Experiment 7

Experiment 7, the final experiment, was an attempt to remove the creation bottleneck completely, by doing all creation on the local processor, not on a special creation processor. In order to avoid the creation of duplicate objects, the blackboard level object was locked by the KS until a new object was created or an existing match was found. Local creation, on the same processors as the KS or rule, also eliminated the need for Qlisp process closure creation. Cage's use of Qlisp process closures is one of the most expensive features of the the implementation of Cage. This is because the Cage programming model requires that large amounts of data (often the KS definitions) be copied from shared memory into local memory during, for instance, KS invocation. Clearly, there is a trade-off between the high cost of copying shared data into local memory (cache), giving cheap local data access, and the low cost of copying references to shared memory, with the relatively high cost of access to this shared data. A smarter compiler would probably be able to improve on this performance by picking the appropriate shared/copied representation for its data structures. In the case of node creation discussed here, Cage requires the passing of the context of the local processes to the creation handler, which is very expensive. With local creation and without the Qlisp process closure, throughput was improved to 25ms, or a 37% improvement over Experiment 5 (See Figure 9).

Experiment	ms	% over Exp 5
Experiment 5 Single creation processor	40	n/a
Experiment 6 Multiple creation processors	31	22%
Experiment 7 Local creation	25	37%

Figure 9. Throughput Results of Experiments 5, 6, and 7.

4.3.2. Analysis of Speed-up and Throughput Achieved

The Cage experiments resulted in two important measurements. These are the maximum relative speed-up, comparing uniprocessor runs with multiprocessor runs. and the minimum sampling interval, measuring the total throughput.

4.3.2.1. Speed-up

Experiments 1 through 4 resulted in a best speed-up of 5.7x using a 16 processor grid with KSs and rules running concurrently without synchronization. The factors limiting this speed-up include:

- The existence of a central controller
- The serial definition section of KSs
- The inefficient allocation of processes to processors
- The high overhead of data copying during node creation and KS invocation.

The effects of the central controller were minimized in Experiment 2 through the elimination of synchronization at that level. The definitions, which are the local bindings done at the beginning of each KS to maintain data coherence (see Section 2.3), are the only part of the

KS still executed serially. Executing definitions in parallel is an option in Cage, but because of the cost of blackboard object creation (63% of the average definition time) and the difficulty in writing independent definitions, at most a 15% improvement in speed-up could be expected from concurrent definition execution.

Experiments 4 and 5 showed that careful resource allocation could improve speed-up. We believe that further improvements in speed-up are possible with tailored resource allocation for additional Cage processes. While Experiments 6 and 7 only measured throughput, a preliminary run under similar conditions showed a speed-up of 6. Some of this gain is also due to the elimination of the use of spawned processes for object creation, thus eliminating much unnecessary data copying. However, some Qlisp is still needed to program concurrency in Cage's shared memory architecture.

4.3.2.2. Throughput

The second major result of the Cage experiments is the relatively poor throughput achieved. The minimum sampling rate for Cage is about 9 times slower than that of a similar distributed memory system, Poligon, running the same application. The factors limiting speed-up also limit the throughput. In addition, it should be noted that there was no optimization of Cage or the Elint application, which could improve throughput significantly.

4.4. Experiments on Poligon

In this section we briefly describe the experiments performed on the Poligon system to date. Two applications have been mounted on Poligon: *Elint*, as described above and *ParAble*, a diagnostic application for particle accelerator beam-lines [Selig 87]. The experiments with the Elint application have now been completed, whereas those on the ParAble system are in their infancy, so only the Elint application will be considered here.

The experiments that were performed were intended to measure a number of different aspects of the system's performance:

- The speed-up that the Poligon system could deliver.
- The peak throughput of the system.
- The ability of the system to exploit large knowledge bases.
- The granularity of the system.

Experiments to measure these are described in Section 4.4.1.

4.4.1. Experimental Results

The space available for this paper does not allow a full explanation of the experimental results, so the interested reader is again advised to refer to [Nii 88] for more details. It is hoped that the treatment here will be sufficient to give the gist of what we have learned.

It should be noted here that wherever reference is made to absolute times, these are measured in terms of the performance of the simulated hardware on which the Poligon system

runs [Deiagi 86a]. Each processing element of this machine is of about the performance of a TI Explorer™ II processor.¹

4.4.1.1. Measurement of Speed-up and Throughput

In this experiment two different data sets were used. One was designed to allow the Polygon system only to create one pipe in the solution space, the second allows Polygon to create four pipe-lines; it was four times as dense.² The combination of these two results allows us to do the following:

- Measure the peak throughput for the larger data set.
- Determine the contribution to speed-up due simply to pipe-line parallelism.
- Compare the results from the two data sets so as to be able to get a measure of the ability of the system to exploit parallelism in the source data, i.e. data parallelism.

The results from the two data sets are shown in Figure 10.

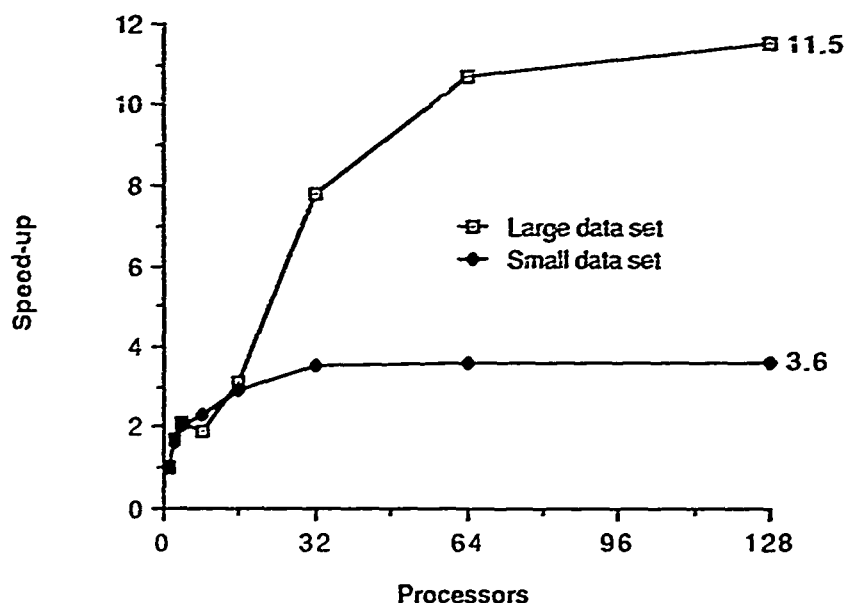


Figure 10. A graph showing the speed-ups derived from the large and small data sets plotted against the number of processors used.

In this experiment, we learned the following:

- The peak speed-up shown in this application due to pipe-line parallelism was 3.6. This showed that although the length of the pipe was three, speed-up was greater than three because of the concurrent execution of rules by the different stages of the pipe.

¹Explorer is a trade mark of Texas Instruments Corporation.

²The small data set can be thought of as representing only one aircraft; the second had four. The data was, therefore, no more complex, there was just more of it.

- The peak throughputs measured from the two data sets were not significantly different. This indicates that Polygon was able to achieve an almost linear increase in speed-up as the problem size of the data set increased, an important result.
- The peak throughput for the system as measured from the larger data set was about 340 μ s per signal data record. Because of the linear increase in performance with data set size it is assumed that with more complex problems higher performance could be achieved. By comparison, the Elint application, when coded to run in the AGE blackboard system took about 3.7 seconds to process each piece of signal data.

4.4.1.2. Measurement of Polygon's Ability to Exploit Large Knowledge Bases

In this experiment the Polygon system was tested using the small data set used above. The Polygon framework was modified so that, whenever a rule was invoked, N rules would be invoked, rather than just one. $N - 1$ of these rules had the special characteristic that they performed almost all of the processing required except for any blackboard modifying updates. This gave a measure of the system load if the knowledge base was N times larger, while still giving the right behavior for this application.

The results from this experiment are shown in Figure 11.

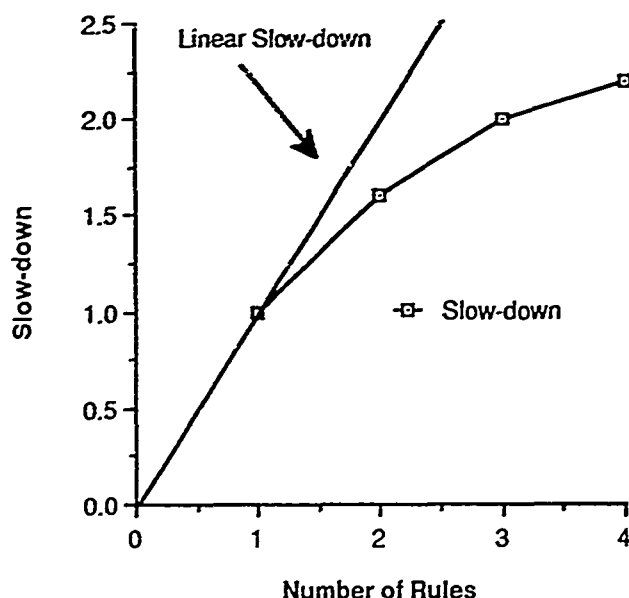


Figure 11. A graph showing application throughput slow-down plotted against the number of rules being fired for each rule-invoking event.

In this experiment, if the system were able to exploit parallelism in the knowledge base to the full, one would expect that the system would not slow down at all as new knowledge was added, i.e. the line shown in Figure 11 would be horizontal. If, on the other hand, the system bogged down completely as more knowledge was added one would expect that the result would be worse than linear slow-down, that is, the plot would appear above the "linear slow-down" line. As can be seen easily from the graph, Polygon's performance was better than linear. In order to perform four times as much work it took only 2.2 times as long. This means that, as long as there are sufficient computational resources, the

Poligon system delivers good performance for a knowledge base whose size is at least up to four times that of the Elint application.¹

4.4.1.3. Measurement of the Granularity of Poligon's Rules

In this experiment some of the internal mechanisms within Poligon were timed in order to get some empirical measure of the granularity of the system.

Within a blackboard system a number of mechanisms are of crucial importance to the performance of the system. Amongst these are slot reads, slot writes, and rule invocation.²

In order to determine the costs of these operations they were performed repeatedly in a manner which allowed the individual costs to be measured with some precision.

The results of these experiments are as follows. It should be noted that all of these results neglect any communication overhead, so they are only representative for local operations.

- Slot reads take $1.36 + 0.94n$ μ s, where n is the number of slots being read at once, Poligon supports a form of multiple slot read operation.
- Slot updates take $18 + 53.7n$ μ s, where n is the number of slots being written. Poligon allows arbitrary user code to be executed during the slot update operation, so this is a representative figure taken from the Elint application. This is for the case of no rules being associated with the slots being updated.
- The overhead cost of starting up a rule's execution is about 1ms per rule invoked.

A substantial part of the time taken performing these operations could be optimized considerably. For instance, a figure of about 500 μ s for rule invocation could relatively easily be achieved in a real system and more than this improvement could be expected for a system which allowed specialized microcode or similar efficiency tuning. This shows that there is a lower bound to the granularity that the user can expect to achieve. For computations taking less than a few milliseconds it may not be worth starting up a rule to perform the computation, the cost of parallel execution would be in excess of the serial execution time.

5. What We Have Learned

We have learned a number of lessons from this project, some of which were counter to our intuitions.

- Our intuition told us that programming a concurrent blackboard system would not be too hard because of the assumed implicit asynchrony in serial blackboard systems. We found this not to be the case. We found the programming task to be difficult and, we believe, a reconceptualization of existing problems will be required in order to port them for efficient parallel execution. The difficulty of implementation of applications is due largely to the divergence of implementations of serial blackboard systems from the pure blackboard model in order to make implementation and programming more manageable as was mentioned in Section 2 and is covered more thoroughly in [Rice 88a].

¹In AGE, the Elint knowledge base was composed of about twenty knowledge sources, each having about three rules.

²Node creation is another important aspect, which was not measured in this experiment.

- We found that the Polygon system and architecture itself performed fairly well. Although programming the system was not trivial, the Polygon system provided a useful abstraction model that allowed the development of an application in a blackboard-like manner that still gave the correct answers and acceptable performance.
- Cage performed reasonably well under the circumstances. We knew from the start that the design of Cage was likely to be limited in the performance that it could deliver because of inherent serialization in the architecture. This, indeed, proved to be the case, but the Cage architecture is still not at all bad for the existing generation of shared-memory multiprocessors.
- We had thought that parallelism in the knowledge base would be crucial to the achievement of high performance. In the applications that we used, knowledge proved to be sparse and the pipe-line parallelism that resulted from it delivered only a factor of three in speed-up. The small amount of speed-up from pipe-line parallelism was due to the short length of the pipes, the lack of applicable knowledge, and the difficulty in balancing the pipes. Most of the parallelism seen in the applications implemented in the Advanced Architectures Project was derived from the data, not the code. The limit to the length of the pipes derived from the application was not one that resulted from the structure of the problem itself, but rather came from the fact that the application was reimplemented for both Polygon and Cage from the AGE implementation, not reformulated.
- When we started the project our intuition told us that the significantly greater cost of communication relative to computation would bias the programmer in favor of doing as much as possible locally before a message was sent. It turned out that doing this increased the granularity of the system and restricted parallelism. We found that, although communication is expensive, as long as data keeps flowing along a pipe the price that is paid is in latency, not in speed-up. The fact that processes are not held up by communication is a result of the non-blocking message sending ability of the hardware. Thus, fine-grained systems are likely to be significant for achieving good performance from large multiprocessors, but the increased latency due to distributing the work could have an adverse effect on real-time performance.
- We learned that the simulation of multiprocessors is expensive. A number of projects are interested simply in the difficulties caused by the asynchronous behavior of concurrent systems. Such projects are able to use a simple model for their implementations on existing hardware. We, on the other hand, wanted to measure the performance of our software on the hardware we were developing precisely in order to refine both our hardware and software designs. This is computationally a very expensive task and has proved to be a major limiting factor on the work that we have done. Having said this, however, it should be noted that we are confident that we have achieved better results and have gained deeper insights than we would have done if we had concentrated on building real hardware. Clearly we could have used existing multiprocessors, but all such existing systems to date have poor programming environments and would not have given us the flexibility that we required in terms of hardware architecture. It is difficult to perform experiments to find new hardware architectures when you are stuck with just one architecture.
- Resource allocation was found to be a significant factor in delivering high performance. The fact that blackboard nodes are often long-lived means that an even load balance can easily be disrupted by a few busy nodes. In the experiments reported here the allocation of processes to processors was done randomly for all of the Polygon experiments and for the early Cage experiments. Other experiments in the Advanced Architectures

Project have shown that, compared to the ideal, perfect load balanced state,¹ even with careful site allocation the Elint application lost 30% in efficiency and delivered 30% less speed-up than in the perfectly load balanced case. This could not be recovered through the use of more processors [Delagi 88].

6. Conclusions

In this paper we have described Poligon, a blackboard framework designed to operate on distributed-memory multiprocessors, and Cage, a blackboard framework for shared-memory machines. We have described experiments performed on both of these systems, shown the results and discussed the conclusions that can be drawn from them, and mentioned some lessons that were learned along the way.

We have shown that the Poligon system can deliver a speed-up for the Elint application of nearly twelve, with near linear speed-up gain with increasing problem complexity. We have also shown significantly better than linear slow-down as a result of increasing knowledge base complexity. We are confident, therefore, that given a larger problem Poligon could deliver significantly more speed-up than this.

Cage has been shown to execute multiple sets of rules, in the form of KSs, concurrently. A speed-up of 4.12 was achieved by the early experiments, improved to 5.7 with optimizations of the resource allocation and 16 processors, and further improved to almost 6 with 32 processors in the last experiment. The use of a central controller to determine which KSs to run in parallel drastically limited the speed-up possible, no matter how many KSs were executed in parallel. The shallow knowledge base of the application limited concurrency at the rule level, but more rules per KS would increase concurrency. Overall, we believe that, with optimization and deeper applications, Cage can be used as a viable concurrent blackboard environment.

From our work we can conclude that data parallelism is likely to be the most important source of parallelism in the foreseeable future, at least until truly huge knowledge bases are developed. This requires that concurrent problem-solving systems should be not only able to exploit data parallelism but be able to do so in a manner which allows the rapid development, easy maintenance and modification of knowledge bases and encourages the development of software that is not brittle when knowledge is added or removed or when the system meets circumstances that were not anticipated by the programmer. Poligon and Cage are possible first steps in this direction.

7. Bibliography

- [Aiello 86] Nelleke Aiello. *User-Directed Control of Parallelism: The Cage System*. Technical Report KSL-86-31, Knowledge Systems Laboratory, Computer Science Department, Stanford University, April 1986.
- [Delagi 86a] Bruce Delagi. *CARE Users Manual*. Technical Report KSL-86-36, Knowledge Systems Laboratory, Computer Science Department, Stanford University, 1986.
- [Delagi 86b] Bruce A Delagi, Nakul P. Saraiya, Gregory T. Byrd. *LAMINA: CARE Applications Interface*. Technical Report KSL-86-76, Knowledge Systems Laboratory, Computer Science Department, Stanford University, 1986.

¹It was possible to measure this because of being able to "cheat" in the processor allocation for the simulator, assuming global knowledge.

- [Delagi 88] Bruce A. Delagi and Nakul P. Saraiya. *ELINT in LAMINA: Application of a Concurrent Object Language*. Technical Report KSL-88-33, Heuristic Programming Project, Computer Science Department, Stanford University, 1988.
- [Engelmore 88] Robert Engelmore and Tony Morgan (eds.) *Blackboard Systems*. Addison-Wesley Publishing Company Inc., Menlo Park 1988.
- [Ensor 85] J. Robert Ensor and John D. Gabbe. *Transactional Blackboards*. Proceedings of the 9th International Joint Conference on Artificial Intelligence: 340-344, 1985.
- [Gabriel 84] Richard P. Gabriel, and John McCarthy. *Queue-based Multi-processing Lisp*. Proceedings of the ACM Symposium on Lisp and Functional Programming: 25-44, August, 1984
- [Lesser 83] Victor R. Lesser and Daniel D. Corkill. *The Distributed Vehicle Monitoring Testbed: A Tools for the Investigation of Distributed Problem Solving Networks*. The AI Magazine, Fall:15-33, 1983.
- [Nii 79] H. Penny Nii and Nelleke Aiello. *AGE: A Knowledge-based Program for Building Knowledge-based Programs*. Proceedings of the 6th International Joint Conference on Artificial Intelligence: 645-655, 1979.
- [Nii 86] H. Penny Nii. *Blackboard Systems*. Technical Report KSL-86-18, Knowledge Systems Laboratory, Computer Science Department, Stanford University, April 1986. Also in AI Magazine, vol. 7-2 and vol. 7-3, 1986.
- [Nii 88] H. Penny Nii, Nelleke Aiello and James Rice. Experiments on Cage and Poligon: Measuring the Performance of Parallel Blackboard Systems. Technical Report KSL-88-66, Knowledge Systems Laboratory, Computer Science Department, Stanford University, October 1988.
- [Rice 86] James Rice. *The Poligon User's Manual*. Knowledge Systems Laboratory, Computer Science Department, Stanford University, 1986.
- [Rice 88a] James Rice. *Problems with Problem-Solving in Parallel: The Poligon System*. Technical Report KSL-88-04, Knowledge Systems Laboratory, Computer Science Department, Stanford University, January 1988. Also in Proceedings of Third International Conference on Supercomputing, May 1988.
- [Rice 88b] James Rice. *The Advanced Architectures Project*. Technical Report KSL-88-71, Knowledge Systems Laboratory, Computer Science Department, Stanford University, January 1988.
- [Selig 87] Lawrence J. Selig. An Expert System using Numerical Simulation and Optimization to find Particle Beam Line Errors. Technical Report KSL-87-36, Heuristic Programming Project, Computer Science Department, Stanford University, 1987.

The Design and Implementation of Poligon, a High-Performance, Concurrent Blackboard System Shell

by
James Rice

(Rice@Sumex-Aim.Stanford.Edu)

**Knowledge Systems Laboratory
Stanford University
701 Welch Road
Palo Alto, CA* 94304**

The author gratefully acknowledges the support of the following funding agencies for this project: DARPA/RADC, under contract F30602-85-C-0012; NASA, under contract number NCC 2-220; Boeing Computer Services, under contract number W-266875.

* California constantly emits neutrons, which strike other materials and make them radioactive. —
Birmingham (Ala) News

Abstract

I started at the top and worked down.

— Orson Welles.

This paper discusses in detail the design and implementation of Poligon, a concurrent blackboard system, documenting our progress and the problem areas we identified in the process of developing it. It also considers the factors that aid and those that limit the performance of blackboard systems in general and of concurrent blackboard systems in particular, relating these factors to the implementation of Poligon.

1. Introduction

Six Hours a-Day the young Students were employed in this Labour; and the Professor shewed me several Volumes in large Folio already collected, of broken Sentences, which he intended to piece together; and out of those rich Materials to give the World a compleat Body of all Arts and Sciences; which however might be still improved, and much expedited, if the Publick would raise a Fund for making and employing five Hundred such Frames in Lagado, and oblige the Managers to contribute in common their several Collections.

— Jonathan Swift, *Gulliver's Travels*,
Chapter 5 of Part III "A Voyage to Laputa"

The Advanced Architectures Project [Rice 88c] has already published a large number of research results, for example [Nii 88b] and [Saraiya 89]. Up to now, however, we have not described the actual *implementation* of the systems that were produced in order to do our research. During this research we identified solutions and potential problem areas for designing future systems in our target area of research, namely concurrent problem-solving systems. It is important to us that we should be able to disseminate the knowledge that we have gained through our experience so that the obstacles we encountered can be avoided by others. Thus this paper not only highlights our positive results, but also attempts to evaluate our approaches and the problems inherent in these systems.

In this paper we describe the design and implementation of Poligon [Rice 86], a concurrent blackboard system [Nii 86].¹ In this section we briefly outline the reasons why we built Poligon, in Section 2 we describe the design and implementation of AGE [Nii 79], perhaps the archetypal, serial blackboard system shell, so as to introduce the discussion of Poligon's design. In Section 3 we briefly consider the implications of the blackboard model for parallel execution. In Section 4 we discuss the design and implementation of Poligon describing in detail its internal representation. Section 5 focuses on the design of Poligon's program support environment. Throughout the paper we attempt evaluate our approaches based on the outcomes of the project, which are summarized in Section 6.

¹It may be of interest to note that the name *Poligon* originated from the system's ability to run in both serial and parallel modes. Names of parallel systems often begin with the letter *p*. *Poligon* combines the Greek word 'Ολιγος (few) and Πολυγονος (producing many, prolific, from which we derive *polygon*, a many-sided object). Following the same pattern, Poligon's non-CARE mode is called Ohgon.

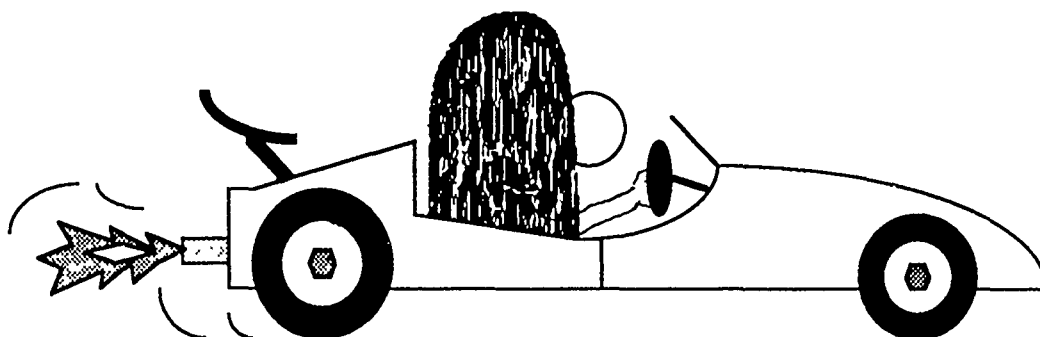
In this paper we assume at least a passing acquaintance with knowledge-based system shells and with concurrency, synchronization, critical sections, and other related concepts. Those less familiar with these issues are directed to [Nii 88a], which provides a thorough explanation of the issues and terminology involved.

This paper discusses not only Poligon, a system that we implemented, and AGE, a system that was implemented a number of years ago but also different ways in which Poligon or some future systems *could* be implemented. We have endeavored to distinguish these subjunctive systems from those that have actually been implemented, but the reader should still be aware that in order to state our beliefs and hypotheses about the future of concurrent blackboard systems we inevitably have to describe how we would implement Poligon in the light of what we have learned or what we would have done if our goals had been different. For instance, some design decisions were based on the goals of flexibility and ease of implementation, whereas if we were to implement again with the primary goal of peak performance, we might choose entirely different design and implementation strategies. The reader should, therefore, note phrases such as "future implementations might...", and "in the best of all possible worlds...", which indicate some impending speculation rather than statement of fact.

1.1. Why High Performance?

Quick is beautiful.

— F.J. Dyson



Eagar likes high-performance machines.

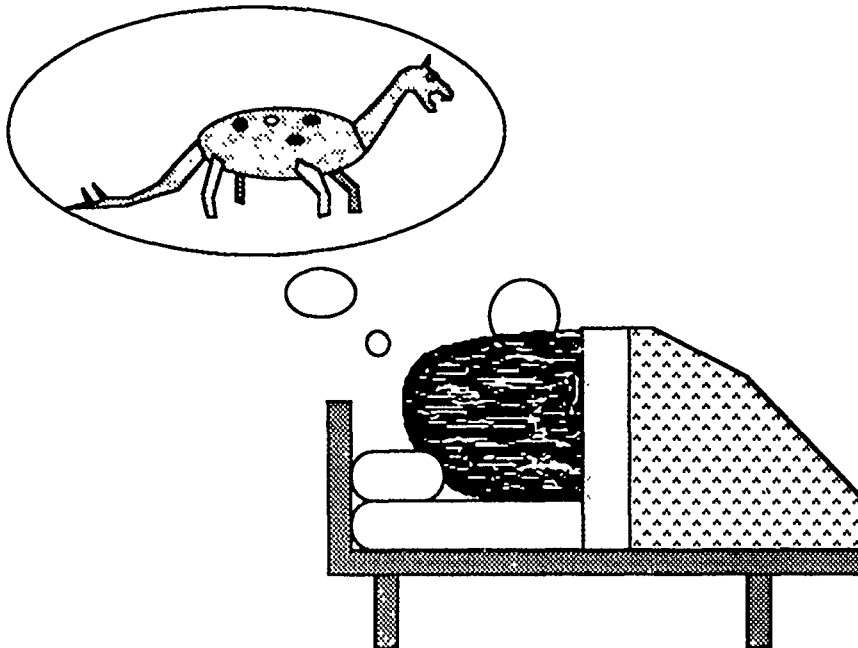
AI research has proceeded for some time without much concern about performance. Researchers have mostly been concerned about the behavior of their programs and were satisfied as long as their programs executed in reasonable time. Now that the technology is maturing and there is increasing pressure to apply AI programming techniques to previously intractable problems in the real world. This inevitably means that the performance of these systems must be compatible with the real world. Hearsay II [Erman 80] provides a good example of this. Even if it had worked perfectly, it would still, at that time, have operated at least ten times too slowly to have been used in the real world. The problem domain we chose to investigate was the interpretation of multiple, continuous signal data streams, such as one might find in radar systems. We already knew this domain to be one in which current blackboard systems have not been able to cope with the performance demands of the real world.

1.2. Why Concurrent?

A physicist had a horseshoe hanging on a door of his laboratory. His colleagues were surprised and asked whether he believed that it would bring luck to his experiments. He answered: "No, I don't believe in superstitions. But I have been told that it works even if you don't believe in it."

— I. B. Cohen

To begin with we should address the question of why we are looking at concurrent systems at all. As mentioned earlier, we need more performance from our AI software in order to apply it to real-world problems. To accomplish our goal, we also need to develop programming methodologies to help us use the evolving generation of machines. It was anticipated that the use of multiple processors could deliver the desired increases in speed. Thus, we wanted parallelism solely in order to gain performance, not to model the physical separation of processors in a distributed, multi-agent system or to improve the reliability of our software.



Eagar likes to be Parallel.

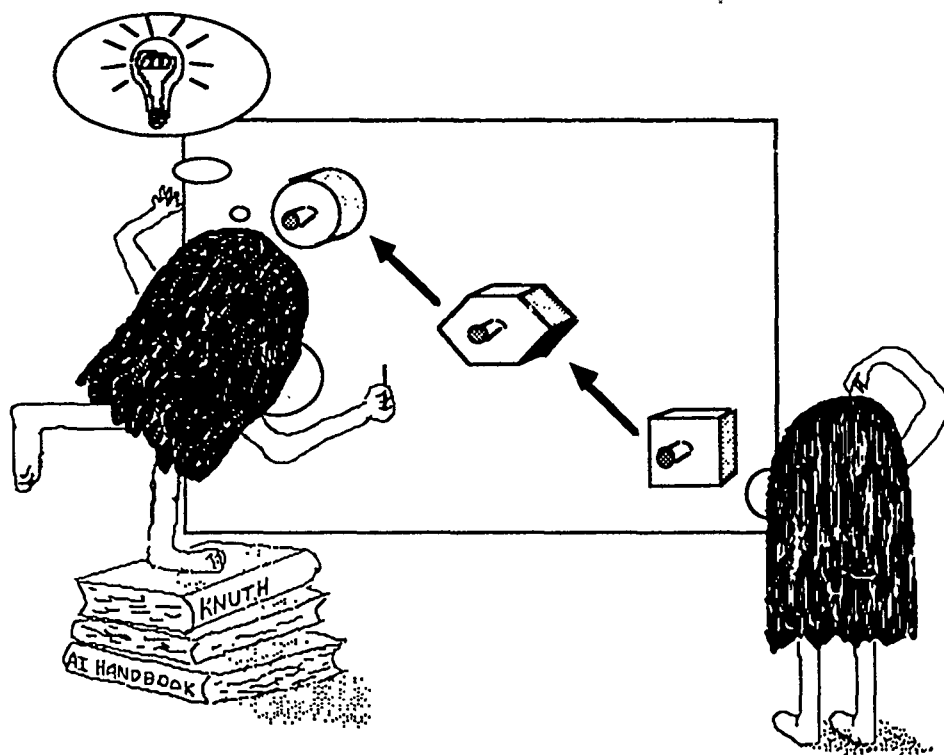
1.3. Why a Blackboard System?

On the atomic bomb: That is the biggest fool thing we have ever done. The bomb will never go off, and I speak as an expert in explosives.

— Admiral William Leahy to President Truman (1945).

Having decided that to investigate concurrent systems, we had to determine which software architectures we were interested in. The primary question was: *Why not write everything in C or assembler with suitable parallelizing directives?* This is by no means a trivial question. The development of any high-level programming tool is based on the hidden assumption that its benefits outweigh its costs. In choosing some form of high-level programming tool over a low-level programming tool, the trade-off is hard to justify when the goal is a high performance system. Except for truly enormous systems, it is generally the

case that software written in assembler is faster and smaller than software written in high-level languages. What one trades off, then, is greater ease of program development, modification, and maintenance against performance. The human cost of software development is great enough that it pays to spend more money on hardware to get the required performance than to spend money on the software being developed.



Eagar finds that a blackboard helps him organize many experts.

Generally this argument applies only in areas of specialized software. Word processing software, like that used to write this paper is usually sold in such quantities that despite the high cost of programming, it is worthwhile to develop new software for existing platforms using less productive methodologies that result in faster program execution. There is, however, a large domain of applications that are only run on a few machines. This software must be developed quickly and modified and maintained easily. Nowhere is this more apparent than in the development of AI software.

Thus, what we are saying is that when we commit ourselves to speeding up expert-system applications using of parallel hardware. We are committed to designing software that can meet its intended purpose. If we elect to design a low-level tool, we accept that it may be hard to use, but it must be very fast. If we design a high-level tool then not only should it be able to solve the problems reasonably quickly, but it should also deliver the benefits that are claimed for high-level tools; modifiability, maintainability, and speed of code development. We were more interested in the design of high-level tools so we were compelled to develop software architectures with the capability to handle the rapid development of concurrent expert systems while still giving high performance. To do this, we sought a computational model around which to develop our design.

At the time, the most promising contender for our prototypical software architecture was the blackboard architecture. Our experience in using this architecture on our project was a significant advantage, but in addition to this, the design itself seemed to admit parallelism through being an intrinsically concurrent problem-solving model. It also seemed to meet our need for a high-level computational model that would help the programmer deal with the complexity of future AI systems. We later learned that blackboard systems are not as parallel as we originally thought; why this is the case is documented in fair detail in [Rice 88a]. Our example suggests, therefore, that one should not pick a programming model for reasons of a superficial match to one's cognitive model of concurrent problem solving. Many of our findings were considerably at variance with our intuition when we started the project. We know of no better architecture than the blackboard model for concurrent problem solving, but this may simply be that few have tried others, other than simple production systems [Gupta 86].

The rest of this paper is biased toward the design of blackboard systems; however, a number of the lessons we learned have broader applicability than just to the field of blackboard systems. Because the blackboard programming model has achieved considerable popularity for reasons independent of its performance, it is quite likely that many will attempt the implementation of concurrent blackboard systems and can benefit from our experience.

2. The Implementation of an Existing Blackboard System - AGE

What we want is a story that starts with an earthquake and works its way up to a climax.

— Samuel Goldwyn

In this section we discuss the design of AGE, a blackboard framework developed at Stanford, both to provide historical and technical background, and to help elucidate the issues involved in developing our project goals.

AGE is a blackboard framework written in Interlisp. It is significant that it is a framework. There have been a number of hard-coded blackboard systems, HASP/SIAP [Nii 82] and Hearsay II [Erman 80] being the best known. We were not interested in the development of hard-coded solutions to our applications since this would have violated the trade-off mentioned in Section 1.3. Having developed the tool, the marginal cost of developing more applications should be relatively small. This is, of course, the same argument that led to the development of compilers.

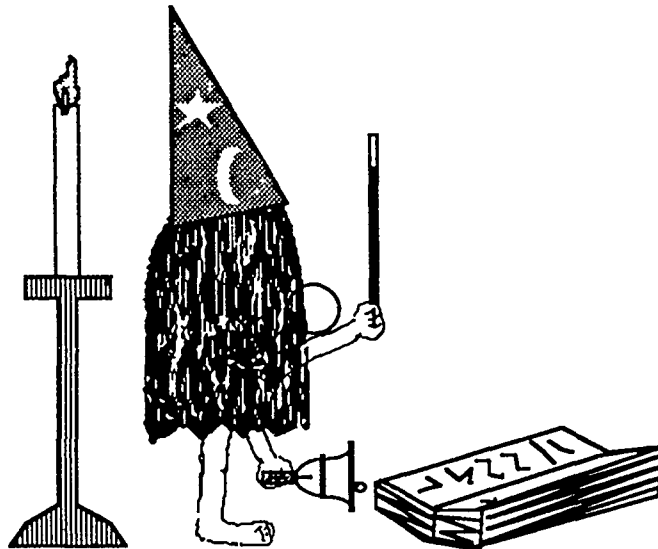
AGE is a system whose design is geared toward the rapid development of blackboard applications. It provides a toolkit for blackboard system development, which contains the infrastructure in which the user's knowledge is to run, as well as such things as rule editors.

2.1. The Blackboard Model

Although the main purpose of this paper is not to explicate the blackboard programming model, it is useful to give a brief description of a canonical blackboard system, in order to show how AGE implements this model. For further information on various blackboard systems the reader may wish to consult [Engelmore 88].

In the blackboard problem-solving model, a group of experts is gathered around a blackboard, each contributing his own knowledge toward solving the problem at hand. The ex-

perts communicate by posting conclusions on the blackboard and watching for other experts posting their conclusions in a similar way. When an expert spots a piece of information that he knows how to handle, he starts working with it. By this means the solution evolves.



Invoking a knowledge source.

This problem-solving model cannot be immediately implemented for a number of reasons, but the primary change that is needed to turn this problem-solving model into a programming model is the inclusion of a scheduling mechanism. This is often referred to as an *opportunistic scheduling scheme* and is often thought to be central to blackboard systems, even though it is only a product of their *implementation*, rather than their *design*. In this case, *opportunistic* means that *the system is sensitive to changes within the evolving solution and, in some manner, tries to invoke the most appropriate piece of knowledge at any given time in order to help the progress of the solution*. This is in contrast to conventional operating system scheduling models in which the scheduler itself has no knowledge of the intent or importance of any given process other than through the use of some "magic" numbers such as priority or quantum numbers. Clearly, a good knowledge-based scheduler ought to be able to use knowledge of the application domain and of the knowledge being executed to find a more responsive and efficient scheduling order.

2.2. AGE, the Canonical Blackboard Shell

Titus Lartius: *Follow Cominius; we must follow you;
Right worthy you priority.*

— Shakespeare, *Coriolanus*, act I scene I.

In this section we discuss the implementation of AGE, highlighting the factors governing its performance.

AGE, being a blackboard system, has a global database that is used to represent the evolving solution – the blackboard. This database is implemented within the native Interlisp environment's heap. The blackboard is made up of a number of data structures that represent the different elements in the solution space. These solution-space elements, called nodes, contain mappings from user-defined names to the values they represent. For instance, a

node might have a slot called *parent*, which has as its associated value the parent of the node in question. These mappings are usually referred to as *attribute/value pairs*.

The knowledge base is composed of a collection of knowledge sources (KSs). These are structures that contain a set of rules that are applied when a knowledge source is invoked. The code for these knowledge sources is also resident within the Lisp system's heap.

A typical blackboard application written in AGE has the following behavior and is shown in Figure 2-1.

- Data coming into the system results in the creation of nodes on the blackboard. These nodes have their slots initialized so that they have some meaningful values in them.
- An event token is passed to the scheduler; in turn the scheduling mechanism invokes the knowledge sources that are interested in that type of event. This involves searching the knowledge base for applicable knowledge sources.
- During the invocation of a knowledge source, computation is performed in order to construct some context relevant to that particular invocation of the knowledge source. The named components of the context are referred to as *knowledge source bindings*. These take the place of local variables in knowledge sources and map local identifiers into computed values. Once these values have been computed, the rules attempt to fire. Rules are implemented as condition/action pairs. If the condition is true, the action or actions are invoked.¹
- Clearly, the evaluation of knowledge source bindings and of any expressions within rules and knowledge sources must be able to look at the nodes on the blackboard. If this were not the case, the knowledge represented by the knowledge source would be unable to do any computation that was dependent on the state of the solution. For this reason, AGE supports a function (called \$Value) that will read the value or values associated with a particular attribute on a particular node. This is AGE's slot read operation.
- Similarly, the knowledge in the system must have some way to record its conclusions. This is done in one of two ways: either the rule that is executing will modify a node or nodes on the blackboard to conform to its new model of reality, or it will create new blackboard nodes to represent new parts of the solution.
- Finally, having performed any appropriate side-effects on the blackboard, the AGE application must perform some action in order to make sure that the system notices the changes that have been made.² This is done by naming the changes with an event token. It is this event token that the scheduling mechanism sees in later system cycles and that causes the subsequent invocation of further knowledge sources.

¹AGE also supports a mechanism for selecting which rules within a given knowledge source are to fire, but this is not germane to our discussion here.

²In the first implementation of MXA [Rice 84], every modification to every slot generated an event. The number of events to be processed grew so large that the application was not able to deal with them reasonably. AGE's strategy of leaving the posting of event tokens to the user is a less automatic approach but more reasonable in practice.

- The system loops around, looking for events on the event queue and processing them in the manner described above. The process of acting on an event and looping around to process the next event is referred to as the *system cycle*.

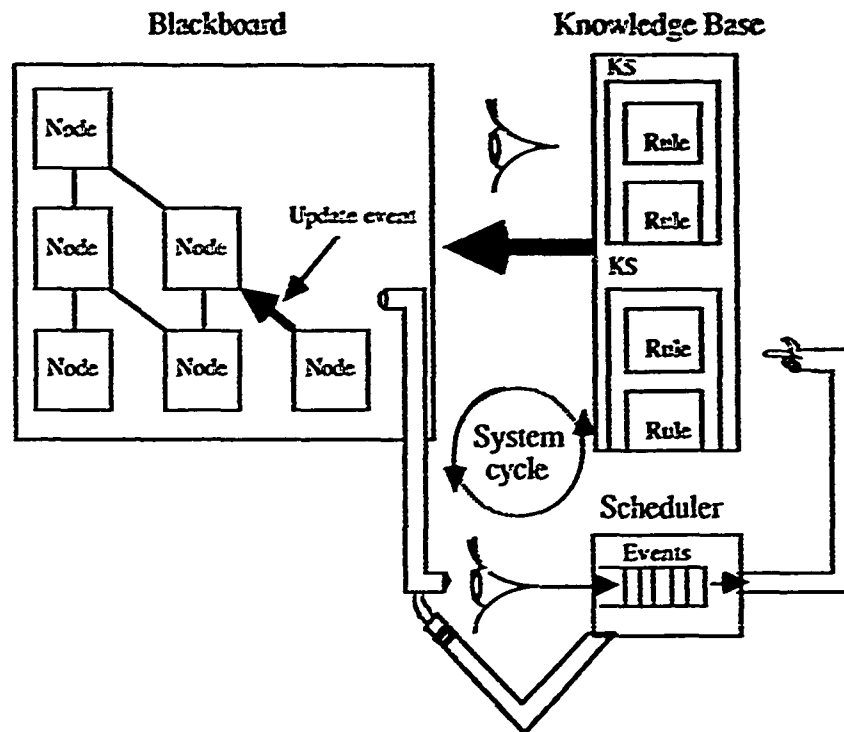


Fig. 2-1. This figure depicts some fundamental aspects of most blackboard systems. A central scheduler sees or is informed of changes on the blackboard, noting them in an event queue. Events are selected from this queue and are used to trigger knowledge sources, which in turn act on the blackboard.

Now that we have outlined the essential run-time behavior of AGE, we can distill from this description the essential components of a blackboard system. These are:

- Dynamic node creation
- Knowledge search — finding applicable knowledge sources for a given event type
- Conflict resolution — Deciding which knowledge source to invoke if more than one is currently invocable
- Knowledge invocation — firing up a knowledge source once it has been selected
- Context evaluation — evaluating any user code necessary for the knowledge source
- Slot reads
- Slot updates — the side-effects that propagate conclusions
- Event posting — recording that something significant has happened

It should also be noted that many blackboard systems have a mechanism for finding nodes that match a certain predicate, AGE's \$Find and MXA's [Rice 84] set creation mechanism are but two examples.

The relative importance of these different aspects of a blackboard system will depend very much on the architecture and on the application for which it is being used. For instance, a system with a large knowledge base of simple rules will stress the knowledge search and invocation mechanisms; an application that does a lot of raw number crunching will require the rapid evaluation of user code. It seems likely that any blackboard system implementation tool (shell) must have some means of performing these tasks, and if it has any aspirations to high performance, a reasonable strategy for making them efficient.¹

AGE was designed primarily as an experimental tool and so was optimized more for program development than run-time performance. We will now discuss the implementation of each component of AGE so that we can contrast its implementation with that of Poligon.

Node creation. AGE nodes are implemented as slots on the property lists of the symbols that name the nodes.² The slots within nodes are represented simply as an AList. Thus, the instantiation of nodes simply involves the creation of the data structure and the recording of it in the level (class) of nodes of the same type. A consequence of this architecture is that nodes of a given level — aircraft, for instance — are only similar by convention. Any node can have a collection of slots that is totally different from another node that is notionally of the same type. This means that space will, in principle, not be wasted in nodes that never use certain slots.

Knowledge search. AGE, like many blackboard systems, offers a user-programmable scheduling mechanism with various precanned strategies. By default events are selected from a global event queue in AGE. Each event encapsulates both the node that caused the event and the event token, which is used to select the applicable knowledge sources in the next cycle. The event token is compared with the preconditions on each knowledge source in the knowledge base, and the set of applicable knowledge sources is delivered. The knowledge source precondition merely has to name the event token against which it is to match. This precondition can be thought of as a filter that helps to select potentially applicable rules.

Conflict resolution. AGE's conflict resolution strategy is extremely simple. If more than one knowledge source could be triggered from an event, then the matching knowledges are fired in the lexical order of their definition.

Knowledge invocation. In AGE, knowledge sources are implemented as list record structures that are interpreted by the scheduling mechanism. The triggering node (the focus node) is taken from the event queue and dynamically bound to a global variable, called *focus.node*. During the invocation of a knowledge source, code executes any knowledge source bindings and then attempts to fire

¹By way of qualification, we should say that at present most serial blackboard systems are optimized for executing either *search* or *recognition* types of applications. It would perhaps be unreasonable to expect any different from a parallel system, though in the best of all possible worlds a blackboard tool would be good at both of these tasks.

²A unique identifier is CONSed to name each node.

the rules by successively testing their conditions and executing their actions, if appropriate.

Context evaluation. In AGE, all user code is interpreted. This means that all knowledge source bindings and all expressions that are evaluated during the execution of rules are also interpreted.

Slot reads. The \$Value function performs Slot reads in a regular manner. It can read the value of slots on any node on the blackboard with cost independent of the node being read. The \$Value function must access the AList within the node structure and must then search for the value named by the slot being accessed. This search must be performed because even if the system were to be compiled it would not be possible to establish at compile-time the location of any given slot within any given class of node.

Node updates. AGE provides a fixed number of system-defined ways to update a node. These allow the modification of the value lists associated with a collection of slots and are performed by calling a procedure (\$Modify or \$Supersede) with a set of arguments that are interpreted so as to find the slots to be updated and the values to put into those slots.

Event posting. Event posting is simple in AGE because of its centralized event queue used by the scheduling mechanism. Whenever an event is to be posted, AGE invokes a procedure that encapsulates the node causing the event and the event token and pushes the event onto the front of the event queue.

Search. Searching a blackboard in a serial system with most implementation techniques is likely to be a linear time operation at best and highly combinatorial at worst. AGE's \$Find operation searches linearly through all the nodes on a blackboard level for a match.

What a high-performance blackboard system should do, therefore, is find ways to make each of these operations fast while preserving the blackboard model.

3. Implications for Parallel Systems

Let's bring it up to date with some snappy nineteenth-century dialogue.

— Samuel Goldwyn

The AGE blackboard model discussed above is based on a number of hidden assumptions that preclude parallel execution. In this section we touch on some of these issues in order to show why certain implementation decisions were made in Poligon.

3.1. The Right Answer

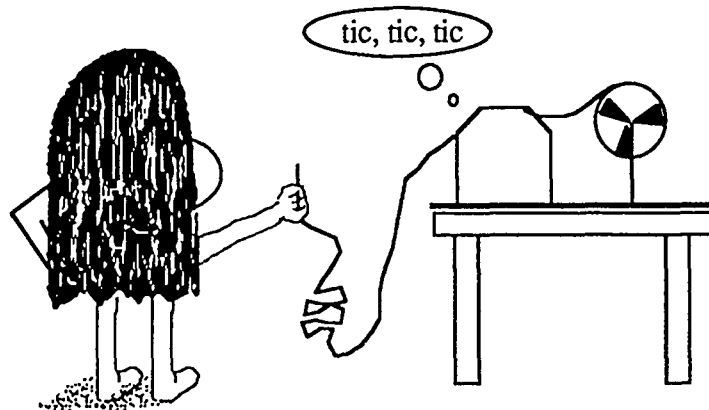
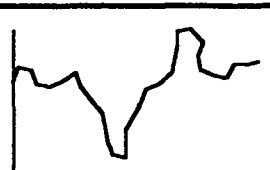
A second order approximation for evaluation of the Köchel (K) numbers of a Mozart symphony (S): $S = 0.27465 + 0.157692K + 0.000159446K^2$

— R.P. and J.R. Cody¹

We should first stress that a concurrent implementation should get the right answer. This is not at all a trivial point and bears some thought. In serial blackboard systems such as AGE, only one thing ever happens at once; in other words, there is no ambiguity about the degree to which the system is converging toward a solution. It is possible to construct a concurrent problem-solving architecture that will act identically to a serial system, but the amount of synchronization required is great enough that the parallel implementation is likely to be slower than the serial version because of the costs of process creation, process switching and synchronization.² As soon as one starts to relax the requirement that a concurrent system should have the same semantics as a serial system, the nondeterminism so introduced can result in a system that either behaves quite unpredictably or fails to converge towards any solution at all.

As a corollary we can say that because a concurrent system allows the simultaneous investigation different avenues that may lead to solutions of differing quality, it is possible to trade off the accuracy of results produced against the overall performance of the system.

Eagar-Jones Average	
Pork Belly Futures	\$4.23
Condo Cave Dwellings Inc.	\$0.03
Wife-Grabber Clubs Corp.	\$9.42
McBrontoburger Intl. (Franchise) Inc.	\$3.11



Eagar finds that timing can be crucial to getting the right answer.

¹This method will give an answer not more than two out, 85 percent of the time.

²This need not be the case if the serial program is, for instance, a pure applicative program, the semantics of which are identical when executed in parallel, but this statement holds true for the parallelization of most current serial programs.

3.2. Instances and Processes

Dogberry: *Come, bind them. Thou naughty varlet!*

— Shakespeare, *Much Ado About Nothing*, act IV, scene II

In concurrent systems it is frequently the case that in order to implement concurrency, each piece of concurrent computation must be executed within a process. For a number of system implementation reasons, these processes are often large and expensive. A serial system need not worry about such matters. Even if such things as dynamic binding are eliminated from the system, process switch time is still likely to be substantially greater than the native system's function call overhead because of the cost of reloading caches. In addition, the cost of processes has a substantial impact on the programming model. This is because an appealing programming model for concurrent computation is that of asynchronously communicating objects. In medium-grain-sized machines, these objects are generally tens to hundreds of bytes in size. Because of page-based stack protection hardware and the increasing size of pages in modern machines, even the minimum size of a stack group is likely to be tens or hundreds of times that of the objects in the system. This means that one cannot sensibly allocate a process to each object without either accepting a huge loss in memory performance or choosing some architecture or computational model that makes more efficient use of stack groups.

3.3. Data Types

Mathematics are a species of Frenchmen; if you say something to them, they translate it into their own language and presto! it is something entirely different.

— Goethe.

Existing serial systems have a well-understood set of data types that are geared toward both the efficient use of existing hardware and the implementation of programmer abstractions. An example of this is structure types, which are often implemented as arrays. Array indexing is fast on all machines. Thus, the user is guaranteed the efficient implementation of his program while preserving the abstraction of naming fields in data items symbolically.

It is not clear yet whether these data structures are appropriate for general concurrent computation, let alone AI programming. CMLisp [Hillis 85] is an example of a language in which new data structures are used to enhance parallelism. Certainly researchers will have to think hard about what data structures are appropriate for concurrent problem solving. Once a reasonable consensus has been reached, we must then convince hardware implementors that these new data types should be supported efficiently in their hardware. Poligon made some steps in this direction, as is mentioned in Section 4.10.

3.4. Control

Control, or *MetaKnowledge* as it is often called, is intrinsically a serializing process, at least as we understand it in the serial blackboard sense. This is because the act of stopping to decide what to do next requires synchronization and then serial processing of the decision process, followed by the serial execution of the knowledge that is selected. Similarly, the knowledge that decides to post events must synchronize on the shared event queue. Strong evidence to support the assertion that control is intrinsically serializing is given in [Nii 88a] and [Aiello 88].

To make efficient use of parallel processors, therefore, a concurrent problem-solving system must try to find ways to avoid the overhead associated with scheduling. As a consequence, system performance will probably degrade because the system is unable to apply the best knowledge all of the time. But given good design, one can at least hope that saving the cost of control and the parallelism extracted as a consequence will buy back by many times the loss in performance caused by executing suboptimal knowledge.

3.5. Hardware

The degree to which our normal serial programs match the hardware on which they run is something we all take for granted. The languages in which we express the programs are themselves biased toward the efficient use of the hardware and vice versa. This is less likely to be the case in the near future. New programming models will have to evolve to cope with new hardware designs, and new programming methodologies will have to be developed. It is clear that a good match between the granularity of the hardware and that of the program will be crucial to the efficient execution of user programs. Likewise, it may well be the case that a good match between programming model and memory architecture will be required. Programming models that use message passing may well be the best application for distributed-memory message-passing hardware. A shared-variable programming methodology may make more efficient use of shared memory machines. This is discussed in [Byrd 88] (see Figure 3-1). An example of a concurrent blackboard system designed to operate on shared-memory machines is Cage [Aiello 86], also part of the Advanced Architectures Project.

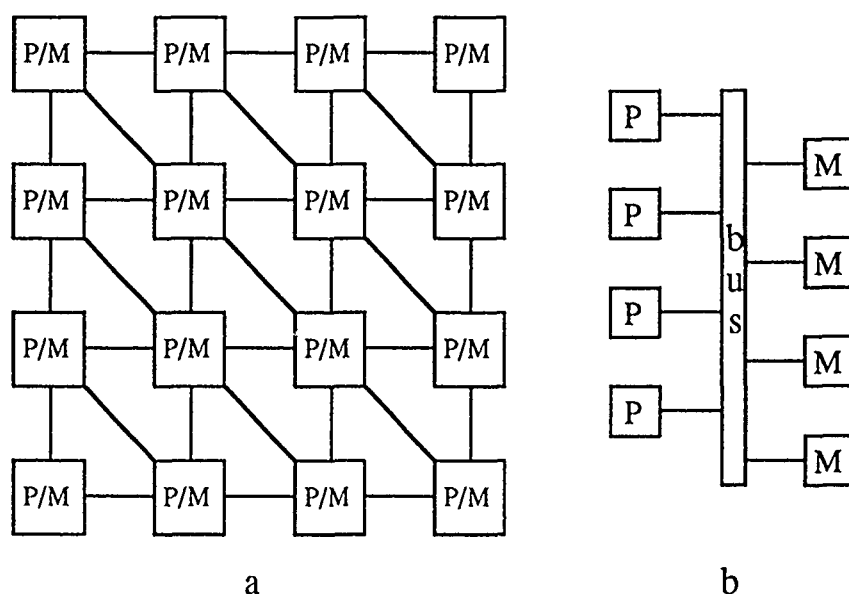


Fig. 3-1. a. A distributed memory machine consists of a collection of processor/memory (P/M) pairs linked by some network — in this case, a six-way connected array. b. A shared memory machine consists of a collection of processors that view a collection of memories as a global resource. In this case, a bus connects the processors to the memories.

3.6. Real-Time

Real-time systems have some special attributes that must affect our way of thinking in a parallel computational environment. Data is likely to arrive out of order if, for instance, network congestion causes unpredictable delays in message transmission. Therefore, pro-

grams must be rugged with respect to data being garbled and must be able to do the "right thing" even when different parts of the program are working at wildly different rates.

4. The Implementation of Poligon

Basic research is what I am doing when I don't know what I am doing.

— Wernher von Braun

In this section we discuss the implementation of Poligon in detail, discussing its history, the problem areas we encountered and, in particular, the areas of a blackboard system mentioned in Section 2 that we believe require improvements in efficiency.

Our initial ideas about Poligon were strongly influenced by the primary objectives of the Advanced Architectures Project. It was broadly assumed that silicon was going to be cheap. We could afford to produce a resource-inefficient design as long as it was usefully faster than one that was more resource efficient. Similarly, we assumed that our hardware design, which was to be run in simulation, would be strongly driven by our evolving programming models. This would allow us to assume the existence of a "blackboard machine" and thus produce designs that could not be efficiently implemented on existing hardware, but that could be implemented on a blackboard machine with suitable hardware or microcode support.¹

These assumptions proved not to be valid simply because of the way the project developed. The hardware design component of the project progressed at a greater rate than the software design, and by the time, some of our problem-solving software began to be implemented, it was clear that we would have to reconsider some of our design decisions in order to get an efficient implementation on the hardware that had been designed. Clearly lack of experience of the real problems of concurrent programming may well have marred our early decisions. The reader is therefore advised to view the following description of the evolving design of Poligon in terms not only of increasing understanding of the underlying problems but also of a gradual appreciation that the targets that we thought were fixed at the beginning of the project were, in fact, moving.

An overriding consideration in the design of Poligon was to develop a system that could, at least in principle, be highly compiled. Existing systems usually have to rely on a great deal of interpretation. Consequently, it was decided early on that if we wanted a feature x to give us the functionality that already exists in serial blackboard systems and there was a similar feature x' that gave similar functionality, but was more highly compilable, we would choose x' over x . This philosophy strongly influenced the designs described in this section.

4.1. The Programming Model

I had a good idea this morning but I didn't like it.

— Samuel Goldwyn

During the initial design of what later became known as Poligon, we decided that we wanted to preserve the abstraction model provided by the blackboard programming

¹It is not at all clear, of course, whether anyone will ever build a dedicated blackboard machine of this type.

metaphor. We already suspected that control would be a significant serializing factor and thought that the communication path between the knowledge base and the blackboard would be a bottleneck if we naïvely parallelized a serial blackboard system. We resolved to produce a design that would eliminate these factors as much as possible, while still retaining the characteristics of a blackboard system.

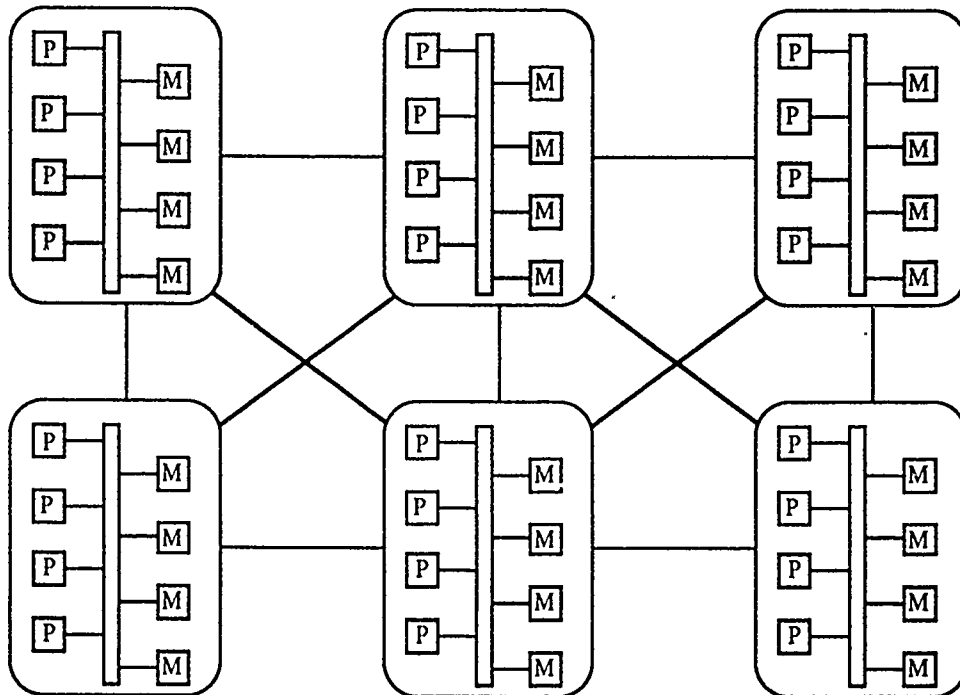


Fig. 4-1. An ideal machine for the Polygon programming model would probably be a collection of shared memory machines linked as if they were a distributed memory machine. This would allow tight coupling and the sharing of data between rule invocations for a particular node and efficient loose coupling between the nodes on the blackboard.

We decided to make the nodes on the blackboard into active agents. A compiler would attach relevant knowledge to the nodes at compile/load time, and the nodes would then invoke their knowledge as daemons triggered by changes to the nodes. This meant that all centralized control would be removed and that all relevant knowledge would have direct access to the data with which it was most concerned. These design ideas were strongly influenced by arguments from our hardware developers that suggested that multiprocessors having very large numbers of processors are most likely to be distributed memory machines. As a result, the cost of reading data from a local processor's memory would probably be much less than that of reading it from a remote processor/memory pair, at least until appropriate new programming models could be developed. It seemed that our own programming model should in some way reflect this asymmetry, though we initially hoped that we could shield the Polygon programmer from this. An idealized machine model for the Polygon programming model is shown in Figure 4-1.

It is important to note that the programming model was strongly influenced by the known implementation model of the CARE machine. This model encouraged a value-passing model of computation, so Polygon was to allow no global variables, and the values transmitted as arguments to any messages sent by the system would be copies of the original values, not remote pointers to the actual values. Remote-Address pointer objects were the only type of pointer that could be transmitted between processing elements. These are

pointers to the streams used to communicate between processes. In the CARE machine, the copying of data is performed by a special processor that handles operating system and communication functions. The user's application is not held up by the copying of message arguments, since this happens in parallel with user code evaluation. Thus, in the following discussion, whenever reference is made to messages being sent or to values being transmitted the reader should remember that these are always copies of the data structures on the originating processor.

Another aspect of the CARE machine model is the semantics of message passing. Unlike the sending of a messages in a Flavors program, for instance, messages in the CARE machine model do not have procedure-call semantics. Returning a value from the computation performed as the result of a message is not mandatory, nor, when a value is returned, will this reply necessarily be sent to the originator of the message. Messages in CARE have explicit *clients*. The *clients* of a message are a collection of the nodes that will need to know the values derived from the computation invoked by the message. This set may be null. Therefore, while much message passing in Poligon has procedure-call semantics this is only because the clients of the messages are often the same as the originators of the messages. This is not always the case, however.

4.2. The Structure of Nodes

On trapping a lion in a desert [Petard 38]: The "Mengentheoretisch" method. *We observe that the desert is a separable space. It therefore contains an enumerable dense set of points, from which can be extracted a sequence having the lion as limit. We then approach the lion stealthily along this sequence, bearing with us suitable equipment.*

The development of Poligon started on SymbolicsTM Lisp Machines¹ and later, upon their arrival, continued on ExplorerTM Lisp Machines.² Because of the strongly object-oriented programming model we envisaged for Poligon, we decided to implement Poligon using the native Flavors system resident on both of these MIT-based Lisp Machines. This decision was motivated primarily by a desire for good performance, compatibility, and good support from the programming environment. Considerable programming effort had already been spent on the development of the CARE simulator [Delagi 88a], which is also written in Flavors. This encouraged us to keep a homogeneous implementation with the underlying simulator.

Poligon nodes, therefore, are implemented as instances of Flavors. We had decided to trade extra compilation effort in favor of higher performance, so we were able to implement slots using Flavors instance variables. We knew that the Flavors model itself would only be adequate as a low-level implementation model. Since the message-passing semantics of Flavors programs are incompatible with the message-passing semantics that we envisaged from the simulated hardware, we had to build a number of layers on top of the Flavors representation of nodes. For consistency the classes of nodes on the blackboard were themselves represented on the blackboard. This was a departure from the AGE model, in which the levels were not really on the blackboard as first-class citizens. In Poligon it was decided that classes should be first-class citizens, and that we should have a general class/metaclass hierarchy in order to describe the complexity of the taxonomy in the prob-

¹Symbolics is a trademark of Symbolics Corporation.

²Explorer is a trademark of Texas Instruments, Inc.

lem domains we envisaged and to implement Polygon's equivalent of class variables. The benefits of multiple compile-time inheritance also seemed worth having in Polygon.

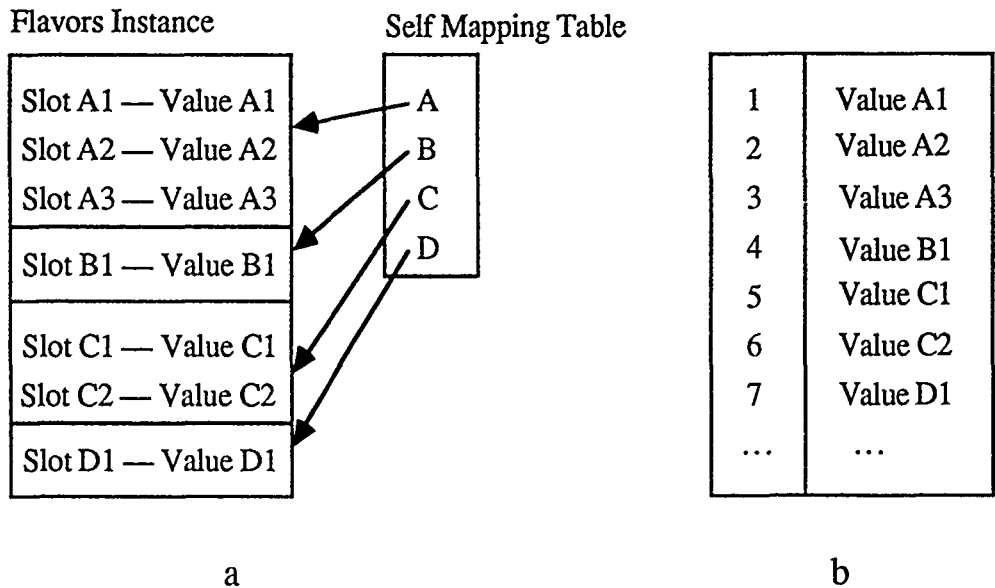


Fig. 4-2. a. The implementation of nodes in Polygon as Flavors instances. To find a slot, the system must indirect through the Self-Mapping Table to find the offset of the component flavor in the instance. b. An ideal implementation of nodes in Polygon would compile all slot references into array indices.

We knew from the start that, in an ideal real-world implementation of the Polygon model, a high performance blackboard system would compile its nodes into arrays, with slot references being compiled into simple array references. This was not done for ease of implementation. In a blackboard system, such as Polygon, however, one can trade off *some* generality for performance and allow optimization through somewhat strong typing and can make a simplifying assumption about the multiple inheritance on the blackboard. If the only classes that are ever instantiated are the leaf classes in the class hierarchy, then knowing the type of a node will always allow the computation of a slot as a fixed offset (see Figure 4-2). This implementation strategy would limit modularity, since it would not allow one to optimize rules that were inherited from abstract classes, but this might be a reasonable assumption in an implementation used in the field. Even without making this assumption, slot access can be effectively optimized given the type of the node in question, so this implementation seemed reasonable. Certainly, Polygon as we implemented it was not as well optimized as this, but at least in principle it could have been.¹

Nodes, therefore, are instances of Flavors. These are composed in a set of class declarations specified by the user. The user now no longer has the ability to associate arbitrary

¹Multiple inheritance can also be supported with fixed position slot access by the use of block compilation and a graph-colouring algorithm to allocate unique slot locations to all of the slots in the class hierarchy that is to be instantiated. Using this strategy, however, instances can easily end up with large "holes" in which slots for unincluded mixins could have been. The optimization of this method so as to minimize the size of these holes is a non trivial problem but can have reasonable solutions for any given application. With this method, data space is traded off against speed, whereas the strategy mentioned above trades off generality against speed.

properties with arbitrary nodes in the solution space, a clear trade-off between run-time performance and space.¹ An example of this composition of classes is shown in Figure 4-3.

```
Class Flying-Thing :  
  Slots :  
  
Class Aircraft :  
  Superclasses : Flying-Thing  
  Slots :  
    Wheels  
    Wings  
  
Class Bird :  
  Superclasses : Flying-Thing  
  Slots : Weight  
  
Class FAA-Controlled-Thing :  
  Slots : Serial-Number  
  
Class Civil-Aircraft :  
  Superclasses : FAA-Controlled-Thing, Aircraft  
  Slots :
```

Fig. 4-3. Some example class declarations for a Polygon program. Birds and aircraft are flying things, and civil aircraft are both generic aircraft and things controlled by the FAA. Classes with names specified after the keyword Slots are the names of slots added by the class, to which they belong.

Nodes in the Polygon model communicate by posting messages to one another. These messages are not seen at the language level. Messages are received in a task queue and are processed one at a time by the nodes to which they were sent. Each node has its own such message queue, which is implemented as a stream (see Figure 4-4). Streams of values are one of the interprocess communications primitives that the CARE architecture supports. Objects running on a CARE machine communicate through these streams, and it is common practice for these objects to have only one stream that receive messages – the *self-stream*. In fact, whenever a Polygon program refers to a node, it is actually referring to the remote address of that node's self-stream. It is these remote addresses that are embedded in user data structures and passed around between nodes and processors.

As mentioned above, it is important for a concurrent programming model to have an efficient method for using stack groups. When we started the development of Polygon we had no such method, believing that nodes would not be all that expensive, and associated a fully fledged process with each one. The Polygon model for reading remote values was based on that of futures [Halstead 84]. This programming model, at least in its general implementation as used by Polygon, assumes that any process can stop at any point in order to wait for the value associated with a future, i.e., in order to perform the defuturing coercion. This may not be a good idea in practice because there can be pathological cases that use up all available memory by creating processes.

¹One could generalize this by allowing behavior like `sl.property-list mixin` as well as fixed position slots, but we had no great interest in doing this for our applications. Clearly, slots of this type could not be as highly optimized as positional slots.

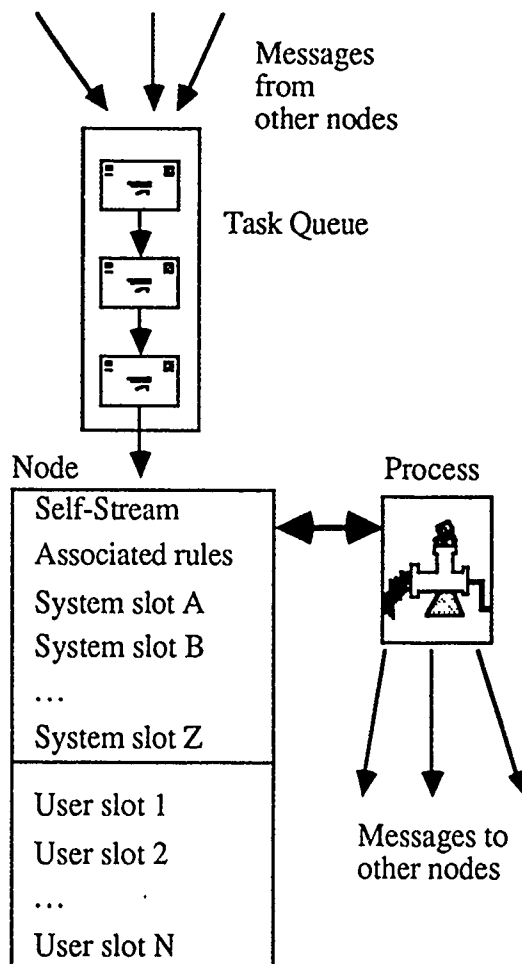


Fig. 4-4. Messages sent by nodes arrive in the node's self-stream. The node's process then processes them one by one, which may result in the sending of other messages.

Other work on the Advanced Architectures Project has developed a different programming model that is implemented in a system called *Lamina* [Delagi 86]. This model has restartable, run-to-completion code fragments. If a process needs a value from a stream that is not available, it aborts itself and restarts when a value arrives on that stream. This means that the process need not hold any state on the stack, so the stack group can be reused even though a process switch has occurred. There is clearly a need to be able to pass state onto the process when it is restarted so as to encapsulate the computation at the point of suspension. This is done by creating a closure that represents the continuation for the computation. The performance trade-off here is between the size of the heap-allocated closure that is CONSed, which will eventually have to be garbage collected, and the stack allocation of state, which is cheap while the stacks themselves are expensive. The programming trade-off is between the user being forced to encapsulate state explicitly and state being recorded automatically.

The advantage of the *Lamina* programming model is that it allows the user to have a good idea of the stack resource requirements of his programs. The disadvantage is that the user has lost the simple procedure-call semantics of a futures-based programming model.

4.3. The Rule-Triggering Mechanism and the Use of Stack Groups

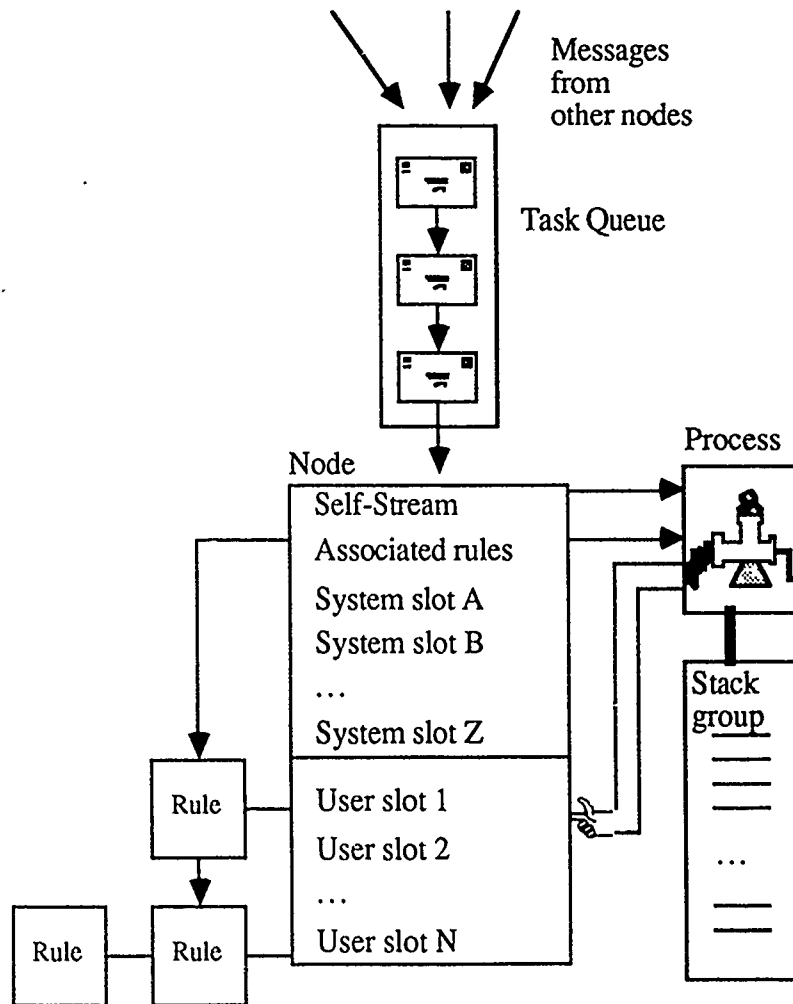


Fig. 4-5. Messages asking for slot reads or updates are collected in a task queue associated with the self-stream of the Polygon node. A process associated with the node reads tasks from the stream and executes them. Slot updates can cause the invocation of rules, which start up in the same process.

In place of a central scheduling mechanism, Polygon provides a daemon-driven mechanism for knowledge activation. We decided at the beginning that we could probably make the system work by triggering the knowledge in the system as daemons on updates to slots. The applications written using Polygon demonstrate that this in fact possible. Unfortunately, project resources did not allow us to test any other, different invocation strategies.¹ As shown in Figure 4-5, updates to nodes, as well as requests to read slot values, arrive in the task queue associated with the self-stream of a node. These are read

¹On a number of occasions, for instance, we were interested in allowing rules to be triggered by more than one slot. This was not implemented because the meaning of this deceptively simple goal is not at all obvious. What do you do when one slot is triggered but not another? Do you go into a partially triggered state and wait until the other slot is triggered? If you do, how long do you wait before you decide that the rule should not fire? All of these issues seemed too hard to tackle when we were investigating so many other new areas. Any new system that is to some extent like Polygon, however, would probably benefit from allowing rules to be triggered by patterns of slot events.

by a process attached to the node, which loops continuously, looking for new things to do. When the process finishes its task, it informs the operating system on its processing element and is suspended, waiting to be reactivated when more work arrives. The operating system has the job of selecting a process from the set of processes with tasks on their self-streams and starting it up.

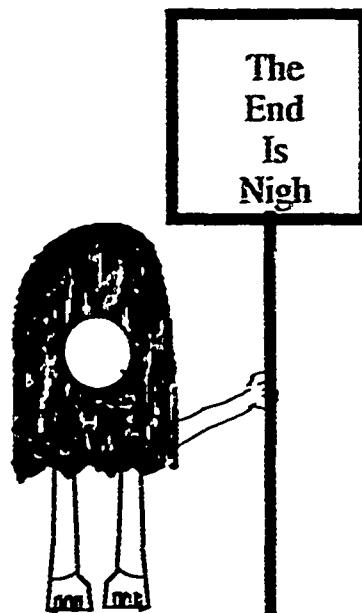
As mentioned above, we determined that using an architecture requiring numerous heavy-weight processes would be a major problem in any Poligon system that we could actually implement, and so we had to rethink the implementation model. We found that by taking advantage of the behavior of the CARE processors' operating system it was in fact possible to find a modified implementation model that would be significantly more efficient.

An important design decision in the CARE machine has been not to allow preemptive scheduling. This greatly improves the efficiency of the operating system and has some significant effects on the semantics of programs that run on a CARE machine. Because all processes communicate by sending messages to each others' self-streams, and because the operating system cannot preempt a process in a random part of the computation by the operating system, other than for fault exceptions, any given process can be sure that the node with which it is associated will not have been changed by another process since it was last activated. This means that in the time between the processing of tasks, the process for the node has unwound to the top level and has no state left on the stack. The process can be switched without the cost of a stack-group switch. This model of computation would, in itself, require the use of only one stack group on any processor, except when the system is intended to do useful work during fault handling. Nevertheless, the number of active stack groups would generally be small.

The problem with this model, which is in effect the Lamina programming model, is that it does not allow the general use of futures. This is because, in the absence of a really smart continuation passing compiler, it is not possible to construct a continuation for every possible point in the program where a future might be defutured, and so futures would not be allowed to be first-class citizens in the source language. They could be used only in very special ways at special times. This did not seem to be consistent with the programming model of Poligon, in which we wanted to preserve the abstraction of procedure-call semantics and a clear source language. For instance, we wanted always to be able to write the expression `«a» + «b»` at any point in the source code, whether or not the expressions `«a»` and `«b»` involved the defuturing of futures. Because in general a programmer simply could not know whether any given piece of data would be a future or would contain futures, we could not expect the user to write complex code to form the continuations for every such case.

What we did in Poligon, therefore, was to try to allow the semantics of generalized futures and yet still try to minimize the number of allocated stack groups. We had originally designed the system on the assumption that a "Poligon machine" would have some sort of hardware or microcode support for trapping access to futures on strict operators, so that the compiler would not have to insert special code for this case. Because of the difficulties of simulating this behavior and the lack of a real Poligon machine, we decided to use the compiler to try to minimize the amount of code needed to check for futures. Through the use of compile-time strictness analysis of the arguments on all functions called in a Poligon application and through the use of type declarations and type propagation, the Poligon compiler is able to deduce areas of code that could not possibly involve defuturing thereby eliminating any code that might have to check for this eventuality. This design had the beneficial property that defuturing would still be a lazy process, but it is still not as efficient as a hardware implementation would be. Thus a Poligon process would block on a future

only at the latest possible moment, allowing the maximum possible time for the future to become satisfied in the background. In fact, futures were often satisfied by the time they were touched, and so a process switch and the need for another stack group was avoided.



Waiting for a Future.

An ideal implementation of the Poligon computational model would be to start up a process to service a message running in a default stack group. This process has no initial state other than the message arguments and the instance variables in the Poligon node. This means that no special initialization or rebinding has to be done to start up the process. This need be only a procedure call. In most cases the processing of a message will not block on a future, so the stack group will unwind back to the top and a new process can be activated without significant cost. In the event that blocking on a future is necessary, a stack-group switch is then performed, swapping out the process and using a new stack group. Stack group switches are, therefore, done lazily.

Poligon's actual implementation of the equivalent behavior is not done in the same way, owing to the CARE simulator's design. CARE distinguishes between fully fledged processes that can be suspended and those that can only be restarted. It does not allow the user to decide halfway through a computation that a process is going to be suspendable. The cost of suspendable processes in CARE is quite a bit higher than that of the lightweight restartable processes. We therefore implemented a scheme whereby the process that actually reads the tasks from the self-stream of the Poligon node is lightweight and restartable one. On the basis of the arguments to the message it has been passed the process tries to prove to itself that a message can be handled without the need to suspend the process. It does so on the basis of the arguments in the message and information that the compiler deduced about the code fragments to be executed. The process is frequently able to deduce that the message can be handled without blocking and so it simply executes the task. If it cannot prove that the message can be handled without blocking it then has to make sure that the message is handled in a fully fledged process. This is done by acquiring a process from a resource of free, suspendable processes and then sending it the same message that was read from the task stream. The restartable process then suspends itself to wait until a reply comes back from the server process. It does this by suspending itself and waiting, not on the self-stream that it normally waits on, but on a stream that is private to these two processes. Thus, the lightweight process that serves the Poligon node can be sure that

nothing will be done to the node until the server process returns. This entire procedure is shown in Figure 4-6.

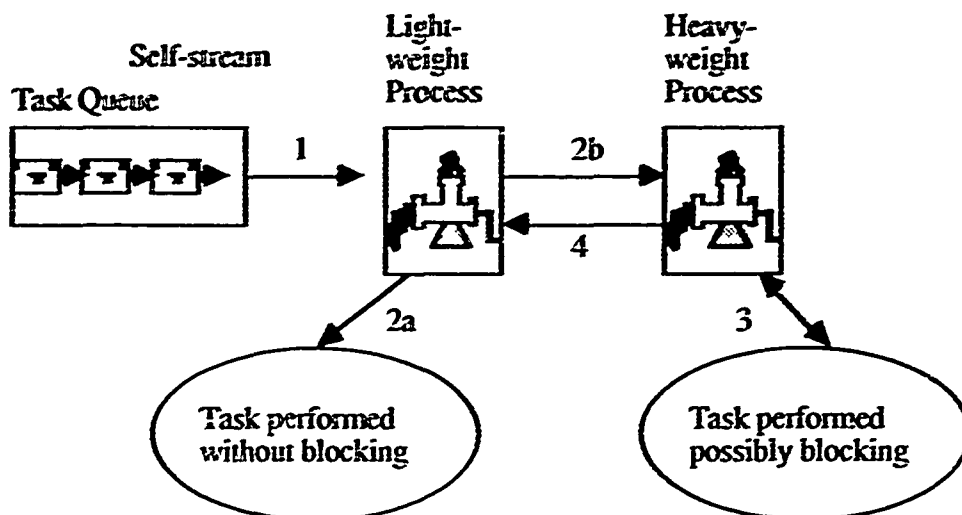


Fig. 4-6. The implementation of Poligon's process model. 1. A message arrives on the self-stream. 2a. If the message can be handled without blocking, it is processed immediately. 2b. If the message may possibly block, a heavyweight process is allocated and told to process the initial message. 3. The original message is processed, possibly suspending itself to wait for futures. 4. The server process replies to the node's process by a private stream.

This implementation model is considerably more expensive than a model that one would use in a production quality system. It involves the cost of trying to deduce whether a process can be handled without blocking, the cost of allocating the server process, the cost of sending the message to the server process, the cost of performing the stack-group switch to the server process and back again, and the cost of servicing the message that contains the reply from the server process. One would not implement such a model on a real machine in the field.

In retrospect, this design seemed to work reasonably well. Empirically the number of stack groups that were ever active was generally much lower than the number of Poligon nodes, and was generally a few times the number of processing elements in the system being used. This was not always the case, however. Occasionally a system would become very backed up because of real-time demands or pathological load-balance problems. As a result, a large number of processors had processes blocked, waiting for replies to messages sent to one processor that was too heavily loaded to service all the requests. It is clear that the model used by Poligon breaks down in such a case. Although it still gives the right answers eventually when all pending futures eventually receive the values they are waiting for, the model does not degrade as gracefully as one would like in instances of poor load balance.

4.4. Reading from Slots

Dogberry: Come hither, neighbour Seacoal. God hath blessed you with a good name: to be a well-favoured man is the gift of fortune; but to write and read comes by nature.

— Shakespeare, *Much Ado About Nothing*. act III scene III

The slot read operation in AGE, as mentioned above, was implemented as a function that used Assoc to find the matching slot being sought in the slot AList of the blackboard node. In Poligon we decided to use positional slots in order to achieve optimum performance.

In fact, we attempted a number of different implementations for Poligon's slots and the means of reading them. This was done as we learned more about the process of problem solving in parallel.

4.4.1. The First Implementation of Slots

The initial implementation of slots in Poligon nodes was simply as lists of values (see Figure 4-7). The user defined the slots that a node would possess in a set of class declarations, which were compiled to produce a suitable set of Flavors for the nodes on the blackboard. For instance, the user could say the following:

```
Class Aircraft :  
  Slots :  
    Wings  
    Wheels
```

This would define a class called Aircraft, all of whose instances would have two user-defined slots, one called Wings and another called Wheels. Similar syntax was used in order to define metaclasses and to mix different superclasses together to implement more complex classes.

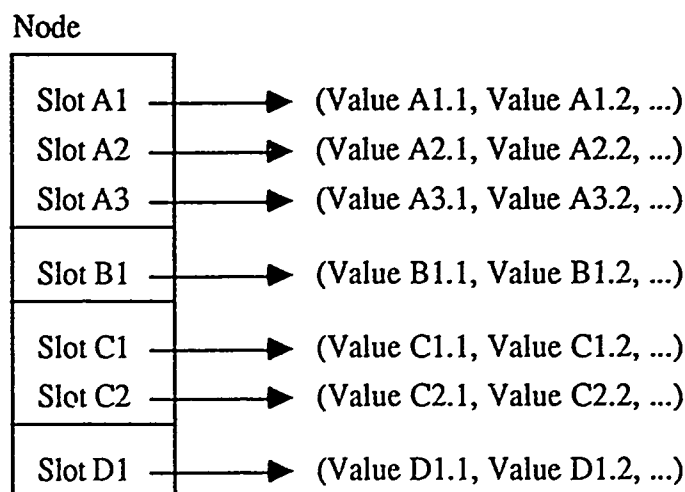


Fig. 4-7. The first implementation of slots for Poligon nodes was simply as value lists.

Operators in the Poligon language allow the user to read the values from a slot. For instance, `foo•wings` would read the first value from the `wings` slot of a node denoted by `foo`, and `foo@wheels` would deliver the list of all `wheels` associated with `foo`.

We quickly found that this was not sufficient. Even though the user could define operators to implement different functionality, because we wanted to support real-time systems in Poligon, we needed some sort of support for timestamping. Data would arrive out of order, and we needed some way ensure that the system would not get confused by the value lists of slots not being in strict temporal sequence.

4.4.2. The Second Implementation of Slots

The next implementation involved a form of automatic time-stamp propagation. Each element in the value list of the slot was encapsulated within a data structure that also contained a timestamp for that value. This is shown in Figure 4-8. These timestamps were set when the data entered the system and were propagated throughout the blackboard during problem solving. When the user evaluated an expression, for example, `«a» + «b»`, and stored the result in a slot, the new value would be timestamped with the time at which the actual computation of the expression `«a» + «b»` finished. Thus, the system could always associate a time with every slot value. To use these timestamps, we introduced a number of new operators. Whereas previously an operator such as `"•"` would simply read the first value from the value list the new operator `"•↑"` would sort all the values in the slot if they needed to be sorted and would then return the most recent value according to the timestamps associated with the values in the slot.

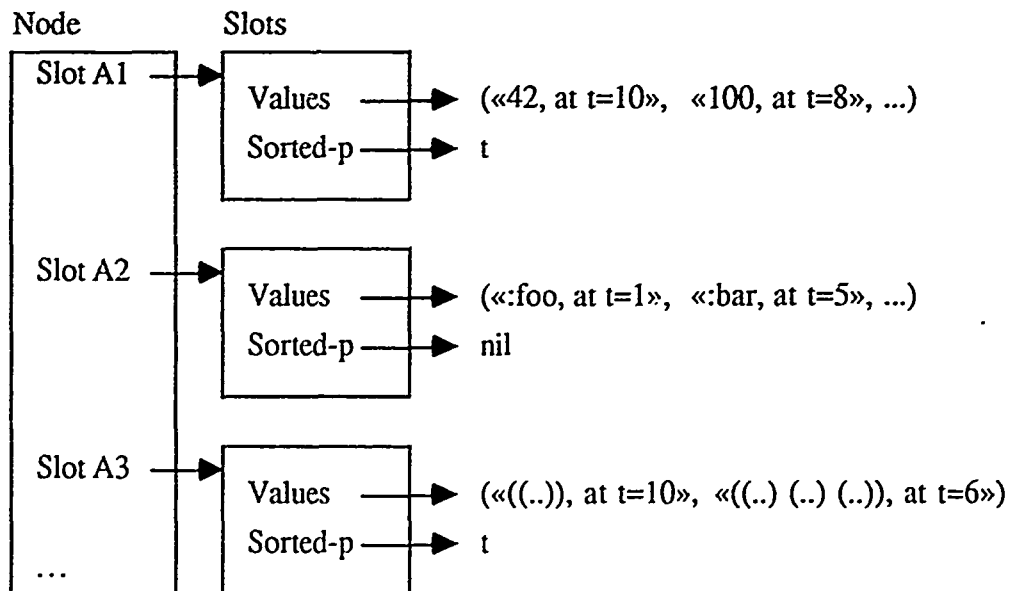


Fig. 4-8. The second implementation of slots for Poligon nodes caused each slot to contain a slot object that had a list of values and a flag that indicated whether or not the values were sorted.

We also found early on that the user often read a number of slots from the same node throughout the body of a rule. This was problematic, because an expression such as `foo•wings = foo•wings` might not be true if the value of the `wings` slot was modified by the time the second read was performed. We needed a way to capture a consistent view of blackboard nodes.

This was addressed by the implementation of a block read construct. Since the program was already sending a message to ask for the value of one slot, the marginal cost of asking for a number of slots at the same time was reasonably low. In Poligon it therefore became possible to write expressions such as `foo&•wings&•wheels` in order to read the values of the `wings` and `wheels` slot within the same critical section. This formalism proved to be very useful and was used in all later designs of slot-reading behavior.

The second slot implementation mechanism worked reasonably well but we found that it was rather expensive and suffered from some major flaws. It was frequently the case that the user's data already had timestamps of its own, which were often at variance with the timestamps that the system had imposed. In addition the user often preferred that the values be sorted on a basis other than the time in the timestamps. This led to the inclusion of an operator that allowed the user to force any value into the system timestamp slot whether it represented a time or not. A secondary flaw was that the user often wanted to index the data in a slot. For instance, when a rule was triggered and was interested in something that happened at time t , it would frequently want to know about other things that happened at time t , or perhaps about things that happened at $t-1$. Getting data on the basis of such an association was not simple to do using the simple block read mechanism described above, since it involved getting all of the values in the slot, possibly from a remote location, and then searching them for the data required.

4.4.3. The Final Implementation of Slots

In developing our third approach we decided that operations like sorting and indexing were fundamentally important to the ease of programming a Poligon application, but that the system should not impose any unreasonable restrictions on the things that could be sorted or the things that could be indexed. It was therefore decided that these operations should be user defined, but also that Poligon should provide the user with some sophisticated and abstract mechanisms for the expression of his program.

Each slot was implemented as an object. In fact, a different Flavor was created for each slot of each class in the system. This allowed the user to specify behavior in a highly focused, per-slot manner, that is, each slot could have specialized behavior associated with it. For example, the user could specify that the values of a slot were to be sorted by a particular predicate, or that they were to be accessed as mappings from indices to values, using a particular index function, or both sorted and indexed. Thus the user could write the following code:

```
Class Aircraft :  
  Fields :  
    Wings :  
      IndexedBy : 'Wing-Side  
    Wheels :  
      SortedBy : '<  
      KeyedBy : 'Tyre-Size
```

As before, all instances of the class `Aircraft` will have two slots, but in this case it is possible to find a wing in the slot `Wings` by looking up which side the wing is on. It is also possible to view the `Wheels` slot as a sorted list of wheels, which is sorted according to the tyre size of those wheels (see Figure 4-9). The "`•`" operator mentioned earlier was modified to allow the specification of an index, so the user could get the left wing of a node `foo` with the expression `foo•wings At :Left`, and could get the smallest wheel with just the expression `foo•wheels`.

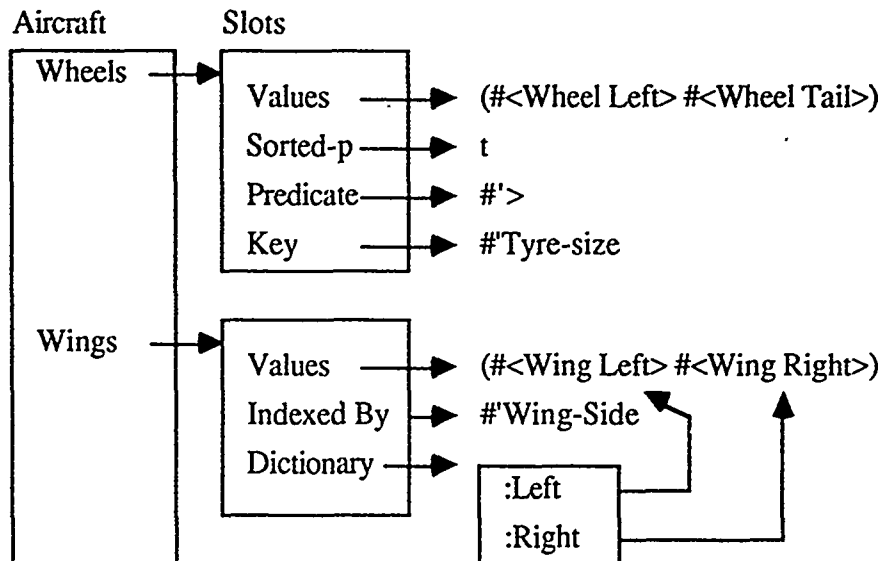


Fig. 4-9. The final implementation of slots for Poligon nodes made each slot point to a slot object, which was of a specialized type unique to that slot. Wings can be seen to have dictionary-like behavior, and Wheels are sorted according to the size of their tyres.

This implementation was more expensive than is strictly necessary. Suitable compilation could expand the state associated with the slot objects into the node itself and could reduce an access simply to an array offset. This would not be in any way incompatible with the compilation of slot accesses into fixed-position array accesses. Similarly, the methods associated with the operations supported by each of these slot types could be fully compiled. In fact, there is no particular need to use methods, in the Flavors sense, since the combined methods for each slot are known at compile-time and no complex method combination is required. It seems likely, therefore, that although this is a considerably more expensive implementation than the original AGE use of a value list, it could be made reasonably efficient and, more important, is considerably more useful in a parallel environment.

4.5. Writing to Slots

The implementation of writing to slots in many ways mirrored the implementation of reading them. This is by no means surprising. The major difference between the evolution of the slot write mechanism as opposed to that of reading slots was that the ability to write multiple slots at the same time was in Poligon from the start. We did this because the same was also the case in other blackboard systems like AGE.

4.5.1. The First Implementation of Slot Updates

As mentioned previously, Poligon's slots originally had no structure to speak of; there was nothing particularly special about the slot-updating process. AGE supported a pair of frequently used slot update procedures called \$Modify and \$Supersede. \$Modify tacked a new element onto the front of the value list of the slot being side-effected, and \$Supersede had the effect of overwrote all the elements in the value list. In effect, \$Modify was an optimization for a frequently used case, and \$Supersede was an implementation of the general case. It was possible to read all the values of the slot, perform an arbitrary transformation on them, and then write out a new value list at the end of the rule's execution.

We did not think that this would be sufficient in Poligon, so we implemented the functionality of \$Modify and \$Supersede as slot update operators, which could be user defined.

The result was that these operators, at least with their AGE semantics, were almost useless. For the reasons already described, things arrived in the slots out of sequence. The Poligon equivalent of the \$Modify operation, which mirrored the "." operator for slot reads, would not really do the "right thing" because one would prefer that the value be put into a more specific place than just the front of the value list. This was due to the fact that the value lists were implicitly time ordered. The Poligon equivalent of the \$Supersede operator proved to be almost useless on account of a hidden critical-section assumption in the AGE model. It was not possible to read all the values of a slot and then write a new list back out again because there was no guarantee that the slot had not been side-effected between the time the slot was read and the time it was written. Writing out a new value list would then destroy any new results that were put in by other rules during the computation.¹

4.5.2. The Second Implementation of Slot Updates

When the slot read mechanism was changed to support timestamping (see Section 4.4.2], we then had a way to avoid the problems we had had with slot updates. Because so much more data was now available to allow the programmer to perform more sophisticated slot updates, there was an explosion in the number of Poligon's slot update operators, which would, for example, *remove an element if it was already present* or *add a new element unless it was already present and was not Nil*.

This added considerable complexity to the programming task. The user had to know about the semantics of the operators that the program would use to *read* a slot in order to pick the correct operator that would *write* that slot. A better abstraction was required. We also noticed that many of our slot update operators seemed to be explicitly fault tolerant. They were all trying to do the "right thing" in the event that the shape of the data in the slot was not quite what was expected. This proved to be an important observation, because it allowed us to develop a much more satisfactory programming methodology and then to build our slot update mechanism around it.

4.5.3. The Final Implementation of Slot Updates

All science is either physics or stamp collecting.

— Ernest Rutherford.

Our final implementation of the slot update mechanism used the methodological idea of "smart" slots. As noted earlier, slots had already been modified by the inclusion of some general mechanisms so that they could be read in ways that were highly application specific. New mechanisms were implemented to support much more focused slot update behavior. The user could now express ideas like *remove this element if it is still there*, and *add this element if it is a new one* in a manner that was both declarative and abstracted out into the class declarations. For instance:

¹This assumes that the whole bodies of rules are not executed within critical sections on the nodes that trigger them. This is discussed at length in Section 4.7.


```

Class Aircraft :
  Fields :
    Wings :
      RemoveIf : 'Still-Present
    Wheels :
      InsertIf : 'Not-Present

```

In this example the user defined functions `Still-Present` and `Not-Present` will be called whenever the program attempts to put new values into these slots or to remove them. In fact, Poligon's default behavior is to do reasonable things in such cases, but this serves as a simple example. The crucial point is that the slots of nodes are now expected to be responsible for their own upkeep. They are intended to have evaluation functions that express the intention or purpose of the slot and thus are capable of assessing any update that is requested and deciding whether to perform it, ignore it, or perform some different update. This results in a sort of local hill-climbing behavior, which allows some Poligon applications to iterate toward a globally reasonable solution. These functions are defined and stored in the same per-slot manner that the sorting and indexing functions are for read operations. They are just extra instance variables in the slot objects that represent the slots.

The functions that the user can specify can be arbitrarily complex. This means that the user has the ability to put arbitrarily expensive code into the slot update critical section. This will clearly lock the node for a long time, but it is better to do it slowly and right than quickly and wrong.

4.5.4. Test-and-Set

There is one more point to discuss regarding slot reads and writes: we found it necessary to implement a test-and-set operation. Although Poligon nodes are now responsible for keeping themselves reasonably coherent, that does not help us if we really need to perform some sort of atomic read/write operation, such as one might want when implementing locks or performing accuracy-critical database operations. Without some sort of atomic test-and-set operation, one would not want to use Poligon to implement a bank transaction system.

We therefore implemented such a test and set operation. The user can now express ideas such as the following:

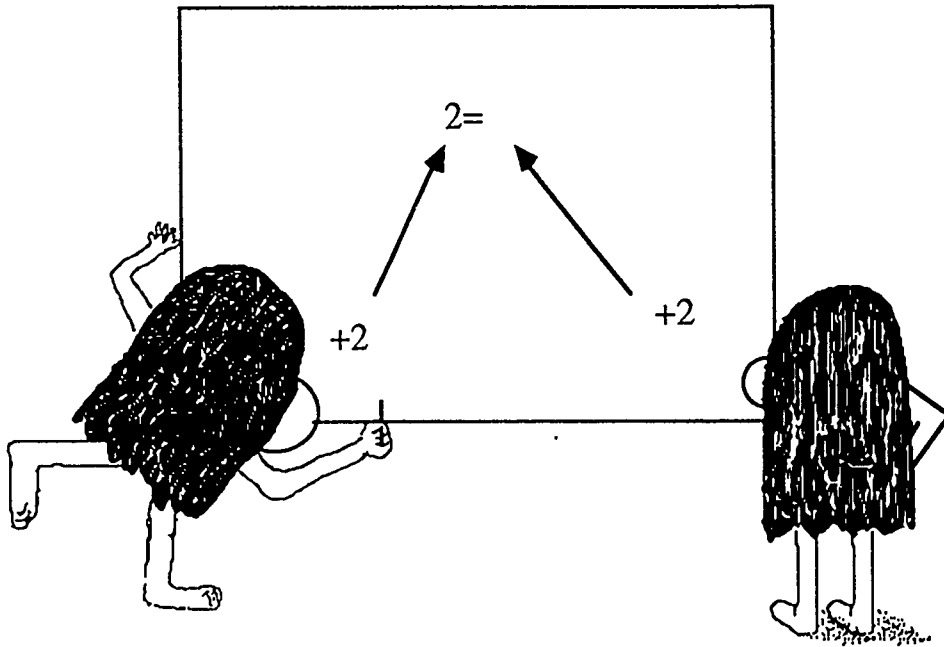
```

foo.wings
  Unless : foo.wings = 2
  Updated Fields :
    wings ← 2

```

In this case, if the node `foo` has two wings then this value is returned; if it doesn't, then `foo` is given two wings and the original number of wings is returned. The test-and-set operation has been used only twice in Poligon applications, but in these cases there didn't seem to be a way of implementing the program without it.

4.6. Creating Instances



Eagar finds that unmanaged instance creation can lead to the wrong answer in a concurrent problem-solving system.

The creation of instances is something that serial systems typically do not handle in a particularly sophisticated manner. The reason for this is that users generally write their knowledge sources so that they are large enough that they know that they are doing the right thing when they create a node. This is, in effect, using large critical sections in order to guarantee that the blackboard is consistent throughout a node creation operation. If knowledge sources are not large, however, especially in processing, it is quite probable that they will create multiple nodes representing the same real-world object. This happens frequently in a parallel blackboard system, and so some mechanism is needed to deal with it.

```
New Instance of Aircraft
Unless :
    Associate(Id-Number, Aircraft@Cache,
              :Return #'Second)
Updated Class Fields :
    Cache ← List(Id-Number, The-Created-Node)
Initialization :
    Wings ← 2
    Wheels ← 3
```

Fig. 4-10. Poligon language source code to create a new instance. If there is an entry in the cache slot of the class node called Aircraft, which is a list of the form ((id <node>) (id <node>)), then the node is returned. If there is no such entry, a new node is created. The new node has its wings and wheels initialized, and the class node's cache slot is updated so that it has an entry for the new node. The node that has just been created is referred to by the name The-Created-Node.

Two options were considered. Either one could manage the creation process so that only the needed nodes are created or one could add extra knowledge so that the system could

reason about the presence of a number of nodes representing the same real object. The latter would, in the general case, require application-specific knowledge in order to achieve this goal, whereas the former could be implemented in a manner that provides a domain independent means of handling node creation. We picked the strategy of managing node creation, knowing that the price would be serialization.

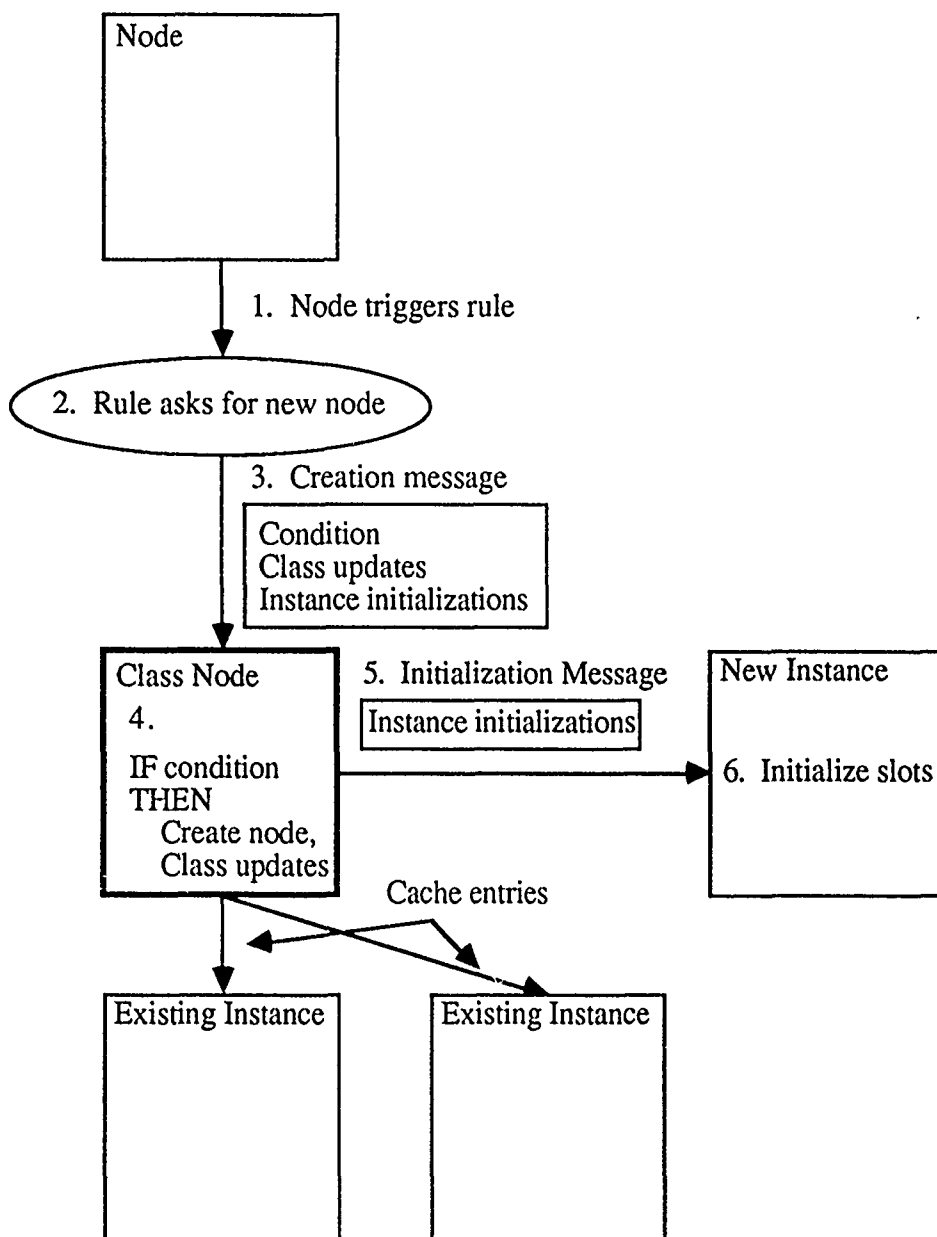


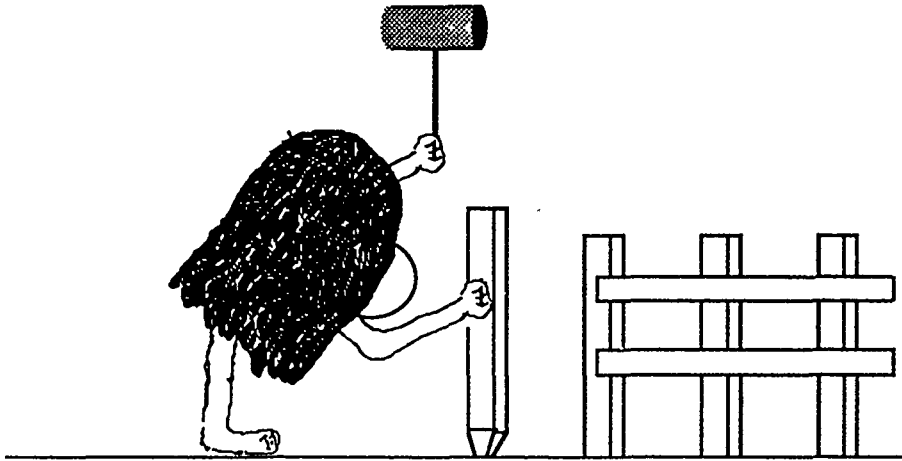
Fig. 4-11. Managed node creation in Polygon. 1. An update to a node triggers a rule. 2. The rule that fires decides that a new instance must be created. 3. A message containing the condition, class update, and initialization closures is sent to the class node for the class to be created. 4. If the condition allows it, the new node is created, the initialization closure is evaluated and passed to the new instance (5), and any class updates are performed. 6. When the initialization closure arrives at the new node, the new node's slots are initialized.

After some experimentation, we developed a fairly complex but general node creation and initialization mechanism for Polygon. We needed to be able to allow the conditional cre-

ation of nodes because, in the general case, a node that represents the thing that we're interested in may already have been created. We needed to be able to construct mappings from identifiers to nodes. These mappings have to allow us to determine whether we already have a node to represent a given real-world object. Finally, we needed to be able to initialize the new node appropriately and make sure that all the right things are executed atomically.

An example piece of Polygon code to create a node is shown in Figure 4-10. The way that this code works is explained below and shown in Figure 4-11.

- First is sends a message to the class node that represents the class of node to be created. This message tells the class node that a new node is to be created or an existing node is to be returned. The message contains three (possibly null) closures as its arguments: a condition closure, a set of class node updates, and an initialization closure for the node to be created. These closures close over the environment of the rule execution so that the program can make reference to expressions in the context of the rule as well as to expressions in the context of the class node or the node being created. Any expressions that require references to be made to anything other than the class node or the node to be created are evaluated before the message is sent. This allows the class and the new node to execute their code without blocking. The expression that asked for the creation of the node returns immediately with a future to the new node. A rule, therefore, need not block in order to create a new node.
- When the message is processed by the class node, the condition part, if supplied, is executed. The condition is executed on the class node because, in the general case, the expressions that make up the condition will want to make reference to caches that are held on the class node.
- If the condition evaluates to `Nil`, a new node is created and entered into the class's list of instances (this process seems reversed, but it actually works.) If the expression is not `Nil` it is taken to denote a preexisting node that represents the solution-space component (node) that we really want.
- If a new node is created, the initialization closure is evaluated. This is a closure that is executed in the context of the class node so that reference can be made to class slots such as the identifier cache. The result of the evaluation of this closure is another closure that is sent to the new node. It is this second closure that actually performs the initialization of the new node's slots. By this point the closure will have picked up all the context it needs from the originating node and the class node. Any rules associated with the slots being initialized will fire for the new node.
- The newly created node is seen by the class node as a future. The class update closure is now invoked with this new node visible. The update closure is therefore able to add the new node to the id cache, even though that node may not yet have been created.



Eagar closes over his environment.

This instance creation mechanism is very general and works reliably, but a number of possibly unnecessary overheads are associated with it. For instance, it may not be necessary to manage the creation of the node. If the node is being created as the result of an unique piece of signal data, the programmer knows that only one node will ever represent this object. Serializing through the class node is unnecessary in this case. In practice, we found that creating nodes to represent low-level objects to represent signal data tended to overload the class node if the instance creation mechanism outlined above was used. Thus, we implemented an optimized form of node creation to allow for this special case. This is shown in Figure 4-12 and works as follows:

- It create an instance directly without reference to the class node and immediately returns with a future to the created node.
- It initializes the node only from the context of the invoking rule, not from the class node.
- It sends a message to the class node telling it that a new node has been created.

Here node creation is unmanaged but faster, but there is another, sometimes undesirable consequence of this operation. Polygon supports operations that can be performed on all instances of a class. Because node creation happens in parallel with the notification of the new node's class, it is possible for the message that notifies the class to be delayed and thus for other pieces of knowledge to execute under the assumption that they are referring to all the instances of a class, but actually they are missing the newly created one. This generally doesn't seem to be much of a problem in Polygon applications, since one usually has to use the full, managed node creation mechanism anyway, in which case the problem doesn't occur, or one's program is generally written so as not to be brittle to this sort of inconsistency.

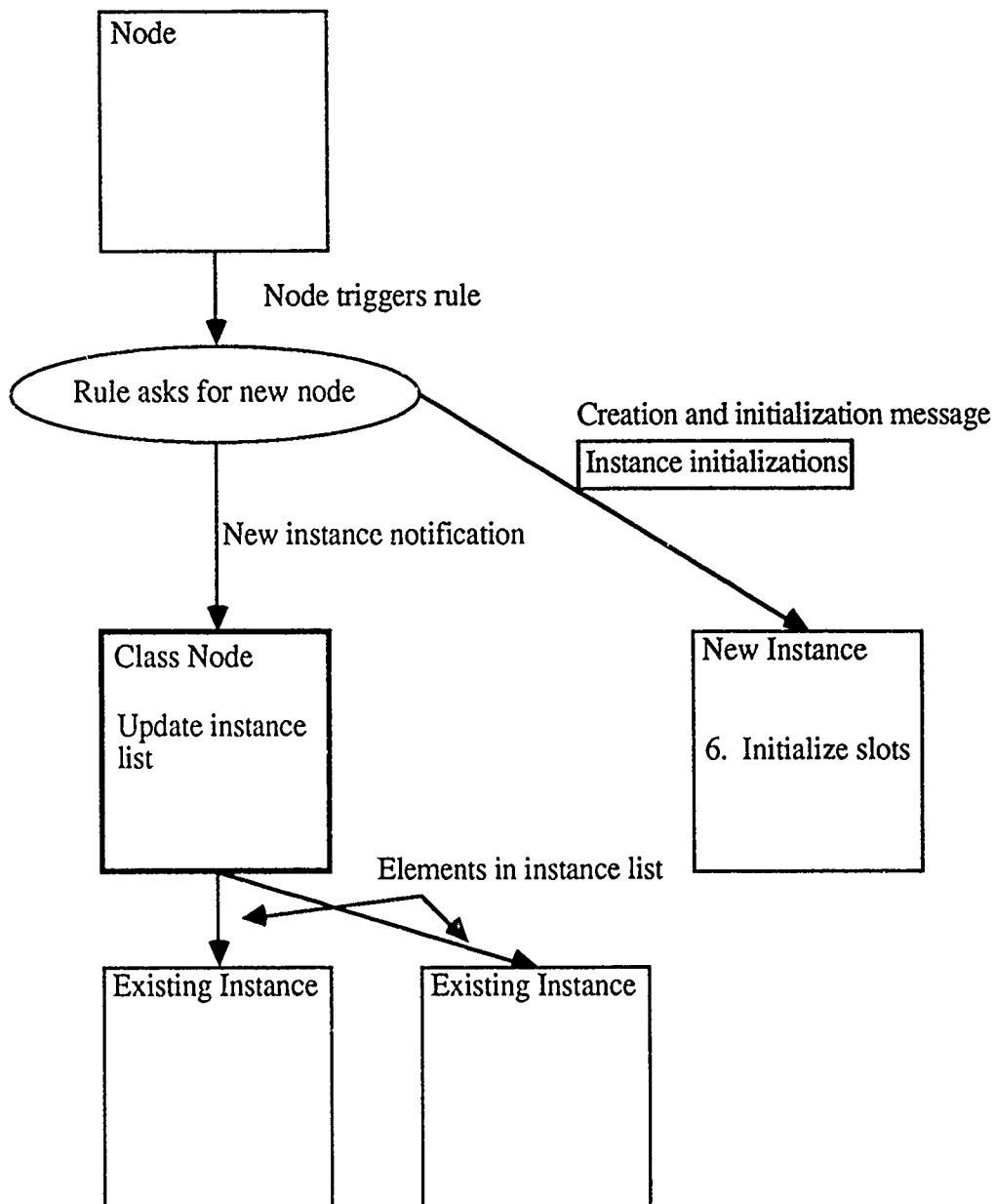


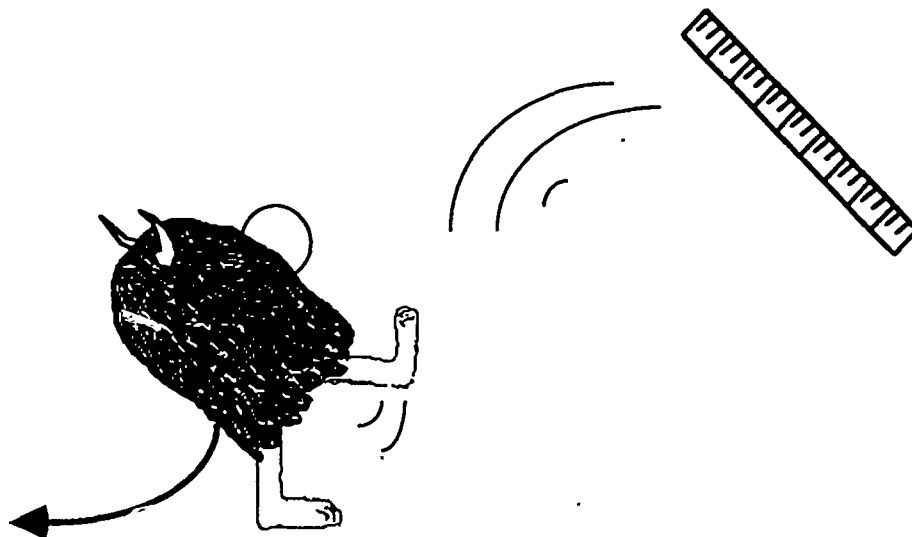
Fig. 4-12. *Optimized node creation. A rule triggered from some node decides that a new instance should be created. The rule invocation creates the instance directly. The class node is notified about the new node by having a future to the new node sent to it and the class node can then add the new node to its instance list.*

4.7. Rule Invocation and the Context of Rule Invocation

Many assumptions about the granularity of the Poligon system were made throughout the development of the system. Perhaps the most significant of these was the decision to allow concurrent rule execution on blackboard nodes. The cost of process creation and/or switching is always going to be significant in the design of a system like Poligon; so the decision that concurrent rule execution should be allowed on each blackboard node necessarily carried with it the assumption that nodes would, in general, have a lot of applicable knowledge at any given time. A second assumption was that the cost of the computation performed by that knowledge would be significantly greater than the cost of rule invocation.

In this section we discuss the implementation of the rule invocation mechanism in Poligon and other related topics.

4.7.1. The Triggering of Rules



A daemon triggers a rule.

As mentioned above, rules are triggered as daemons on the slots of nodes. The slot that triggers a rule is defined by the programmer and is fixed at compile-time.¹ It is not possible, however, simply to fire the rules as soon as the update that would trigger a rule is made. This is because Poligon supports the ability to update a number of slots "simultaneously." To allow rules to be activated before all relevant slot updates had been performed would open the door to very counter-intuitive behavior. Poligon, therefore, collects the significant events on slots as the slots are being updated, and when all of the updates have happened, activates the interested rules.

The evolution of the slot update mechanism was discussed in Section 4.5. A consequence of this implementation is that the system always knows what changes to a given slot were made when it was updated. For instance, if a new wheel were to be added to an aircraft, then the node being updated would remember the node that caused the update and the slot that was updated, the new wheel — or, actually, the set of new wheels, which in this case is a singleton set.²

Once this information has been gathered for the updated slots, the node is free to trigger the associated rules. For this the node must create contexts.

¹This is not the case for expectations, which are described in Section 4.7.4.

²A deficiency of the implementation in Poligon is that there is no way for a rule to recognize whether the values that it is passed, which tell it what caused the rule to fire, are values that were added to or removed from the slot. The rule has to work this out for itself. Clearly this would be a small thing to fix but is worth noting here, since we have found programs that wanted to trigger rules both as a result of inserting an element in a slot and as a result of removing values.

4.7.2. Contexts and Pseudo-Contexts

In AGE, the context in which a rule is executed is that of the knowledge source and the knowledge source bindings. The knowledge source knows only the token that triggered it and the node that caused the triggering.

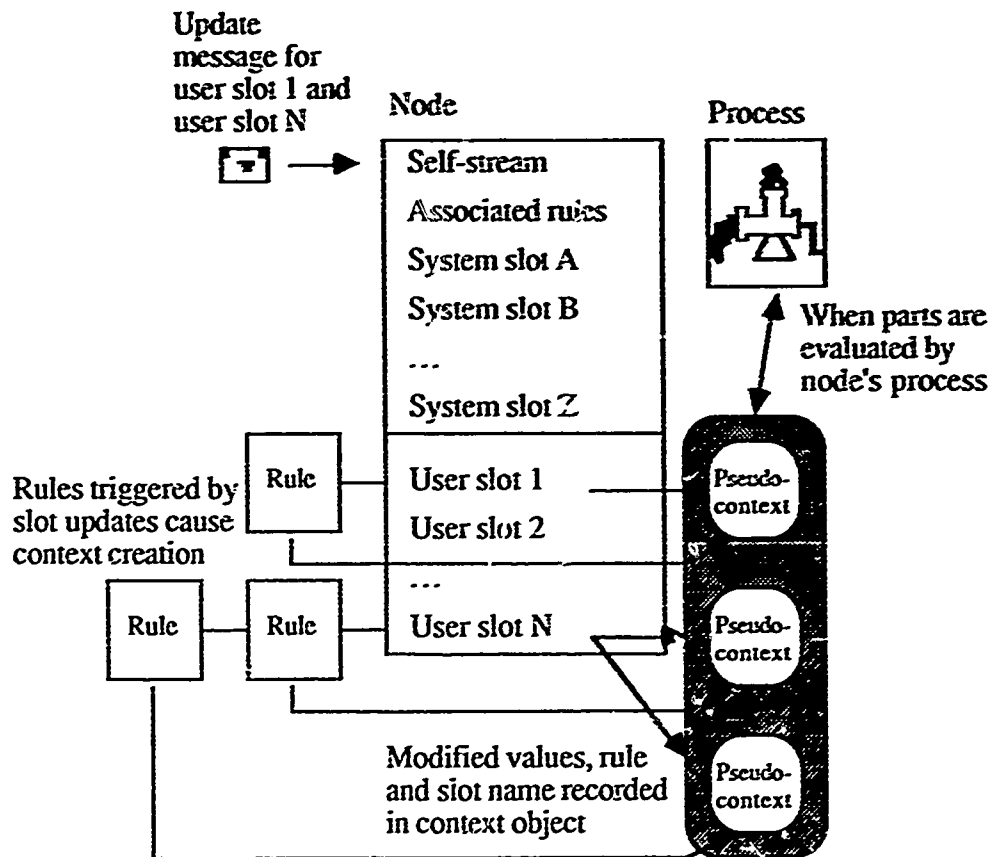


Fig. 4-13. A node receives an update message. The updates, when processed, cause the triggering of the rules watching the slots that were updated. When a rule is triggered, a pseudo-context is created and the When part of the rule is evaluated locally.

In Polygon we decided early on that knowledge sources were too coarse grained to give us the performance we wanted. Consequently, we wanted at the very least to execute rules in parallel; we wanted to compile away the knowledge sources, preventing them from being the scheduling units that they are in serial blackboard systems, and to incorporate into the rule any state that the knowledge source might have had. Polygon has functionality equivalent to knowledge source bindings called *definitions*, which are discussed in Section 4.8. Moreover, a number of existing blackboard systems have a more substantial amount of context when their knowledge sources are activated than was available in AGE. BB1's [Hayes-Roth 85] knowledge source activation records (KSARs) are an example of this. If we were to run our rules in parallel, we knew that we needed some representation of the rule's evaluation context, and that we had to associate a process with each rule in order to execute it. In addition, because Polygon runs on a distributed memory system, each element of which is effectively a uniprocessor, the activations of rules would, in the general case, be running on different processors from that of the node, which caused the activation of the rules. Each rule therefore runs in a different address space from that of the triggering node (see Figure 4-13). The objects that represent the activation of a rule in Polygon are called *contexts*.

Contexts are Flavors instances that contain the following information:

- The triggering node
- The rule being triggered
- The values of the slot that caused the rule to trigger
- The definitions for the rule being executed

Recognizing that the cost of a process switch to a context would be significant and that the reading of values from the focus node would be expensive for a context on a remote processor, we wanted some means of filtering out rule activations before they became too expensive. For this we used a cheap test called the *When* part of a rule. Polygon knowledge sources have no preconditions, since they are compiled away, but rules needed a cheap means of determining whether they were applicable or not. The condition part of a Polygon rule is therefore split into two distinct components: the *When* part and the *If* part. The *When* part is executed by the node for which the rule is triggered. This means that no process switch is performed to evaluate the *When* part and that slot reads to the node can be fast. The *If* part of the rule is executed only as long as the *When* part succeeds and is executed in a different process by a context object. It is therefore useful for the rule to do as much cheap filtering as possible during the *When* part and to perform enough reads to slots on the focus node in order to prevent the context from having to read from the node again if that is possible.

If we were to allow the node to read values from other nodes during the *When* part, the node would have to be able to block until the results came back. This seemed undesirable, because it could cause areas of the evolving solution to lock up for unpredictable periods of time. We thus decided not to allow the *When* parts of rules — or any parts of the user's program that are executed on the nodes themselves — to make remote references. These are only allowed during the *If* or *Action* parts of rules, which are executed by contexts.

In order to be able to evaluate the *When* part of the rule, the node must create a context for the *When* part's execution. We wanted to avoid the cost of process switching during the *When* part, so the node creates an object called a *pseudo-context*. A pseudo-context is just like a context, only it executes within whatever process invokes it. As a result, any state developed during the evaluation of the rule's *When* part is recorded in the pseudo-context. If the *When* part evaluates to true a context is invoked and is passed the state in the pseudo-context as part of its initialization message.

The reason pseudo-context objects are necessary is that the *When* parts of rules can contain references to definitions, which are described in Section 4.8. In retrospect, we can see a justification for having yet another sort of precondition, one that does not allow the use of definitions. The creation of the pseudo-context for the rule, although much cheaper than a process switch to a context, still carries a significant cost. If we could do some prefiltering without this cost, we could expect better performance. Substantial optimization of the pseudo-context creation mechanism could be made, but avoiding this altogether, if possible, seems worthwhile. A smarter compiler might, perhaps, have done some flow analysis on the source program and created the contexts lazily. This would have required a substantial reimplementaion of the rule activation mechanism in Polygon and so was not implemented, but it could well be a useful strategy for any future system built along these lines.

4.7.3. Rule Execution After the When Part is Evaluated

When the When part of a rule is evaluated and is true, the node that started up the rule must invoke a context to process the rest of the rule. Normal rules come in two forms in Poligon: If-Then-(Else), or If-Then-Case-Else. It was found early on that blackboard systems commonly have sets of rules whose form is of the following type:

Rule 1:

```
If «some condition»  
Then «some action»
```

Rule 2:

```
If Not[«some condition»]  
Then «some other action»
```

This is typically not too much of a problem in serial systems because the cost of evaluating such a pair of rules is often reasonably small, but the constraints of a parallel or high-performance system make it desirable to avoid unnecessary rule invocation as much as possible. It is not the cost of the user code that one wants to avoid, since the code will be evaluated nevertheless; it is the evaluation of the system code that creates the contexts for rule invocations and that starts up the rule. In the previous example, it is clear that the two rules are mutually exclusive. They could, therefore, be rewritten in the following form:

Rule 1:

```
If «some condition»  
Then «some action»  
Else «some other action»
```

Poligon supports just such a rule representation and, in fact, generalizes it so that the following rule is possible:

```
Rule Watch-for-changes-in-wheels :  
  Class : Aircraft  
  Slot  : Wheels  
  Condition Part :  
    When : The-Wheels > 0  
    If : The-Aircraft•is-airborne  
    Select :  
      If The-Wheels = 3  
      Then :Land-ok  
      Else :Belly-land  
      EndIf  
  Action Part :  
    :Land-ok :  
      «we have enough wheels to land on the  
      undercarriage»  
    :Belly-land :  
      «do whatever you must to land on your belly»  
  Otherwise Part :  
    «we haven't taken off yet so the change must be  
    due to the maintenance crew»
```

This trivial rule takes advantage of the mutually exclusive execution of its action parts. For any given change in the number of wheels on the plane the rule will be invoked only once.

If we had a number of rules that watched for this change, two rules might start up because the plane did not have enough wheels, but by the time one of them actually came to look at the plane and do something with it, the other rule might have caused more wheels to be added, thus confusing the first rule. The form of rule shown above has proved to be powerful, useful, and efficient.

The Polygon compiler causes the rules that the user expresses to be split up into a large number of functions and methods. The objects representing the rules in the system have a number of slots that refer to the code to execute the different parts of the rules. A very simple piece of code run by the context object looks at the rule object that represents the rule that it is to fire and invokes the relevant parts of the rule as appropriate. This is shown in Figure 4-14.

In designing the Polygon language, we did not want the user to have to worry about picking the parts of the program that would run in parallel and those parts that would run serially. To this effect we designed the language so that the user expressed those parts to be executed serially rather than those to be executed in parallel. The system was then at liberty to execute any other code fragments in parallel if this seemed appropriate. Clearly we wanted to execute as much as possible in parallel, wherever it would be beneficial. Our intention, therefore, was to execute components of the action parts of the rules concurrently and without any synchronization.

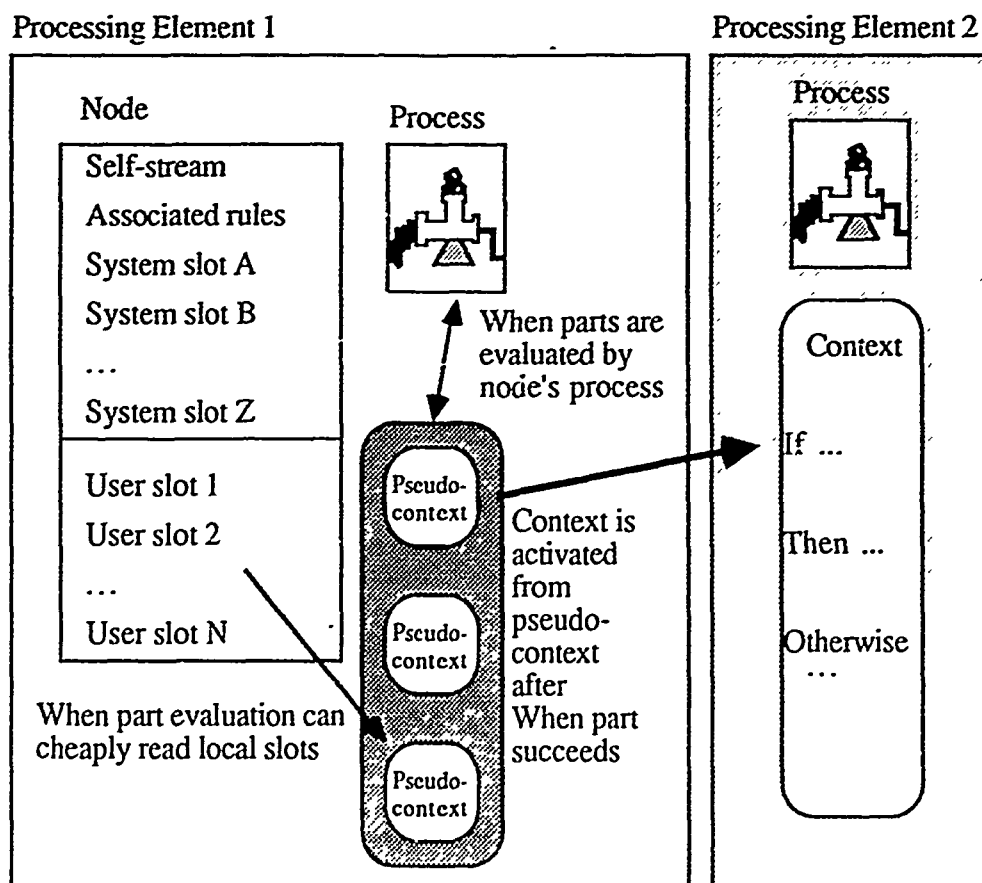


Fig. 4-14. When the When part of a rule evaluates to true, a context object is activated, possibly on another processing element, to evaluate the rest of the rule. Cheap access to local slots can be made during the When part's execution. Slots can be read from the focus node during the evaluation of the rest of the rule but this is discouraged.

The following fragment of Poligon code expresses updates being made to two different nodes:

```
Action Part :  
  Changes :  
    Change Type : Update  
    Updated Node : «aircraft 1»  
    Updated Fields :  
      wheels ← «a new wheel»  
      wings ← «left canard», «right canard»  
  
  Changes :  
    Change Type : Update  
    Updated Node : «aircraft 2»  
    Updated Fields :  
      wheels ← «tail wheel»  
      wings ← «new left wing»
```

We wanted the execution of these two updates not to be held up by each other. What we did was to make the Poligon compiler extract the references to any definitions (see Section 4.8) from the expressions in the action parts and evaluate them whenever possible before entering the actual expressions that perform the updates. In order to perform the updates, the Poligon compiler tries to deduce the best place in which to evaluate the change component. In the cases above, each change will be evaluated on the respective aircraft nodes.¹ A message is sent to the node to do the computation, specifying a method that will be used to make the update and containing a newly CONSed pseudo-context that contains the definitions that are to be used in the evaluation of the action part. The use of a pseudo-context in this case allows a regular mechanism for the evaluation of definitions irrespective of where or when they are evaluated. The compiler has already computed which definitions will be needed in the execution of the action-part clause so only these are sent over. Figure 4-15 illustrates this sort of slot update operation.²

¹There is an assumption here that the code to be executed on these remote nodes will be reasonably cheap. It is left to the user to make sure that expensive expressions are factored out as definitions that are evaluated before the update message is sent.

²This whole design strategy may have been flawed. The creation of pseudo-contexts for the definitions passed to the nodes to perform the action parts is costly. It would probably be better to compile the action part into methods that accepted the definitions as arguments. This, however, would have the consequence that all definitions would definitely have to be evaluated before any part of the action could be executed. As a result, the lazy semantics of the definitions would be lost if any definition references were made in expressions with conditional clauses. This strategy clearly requires more thought.

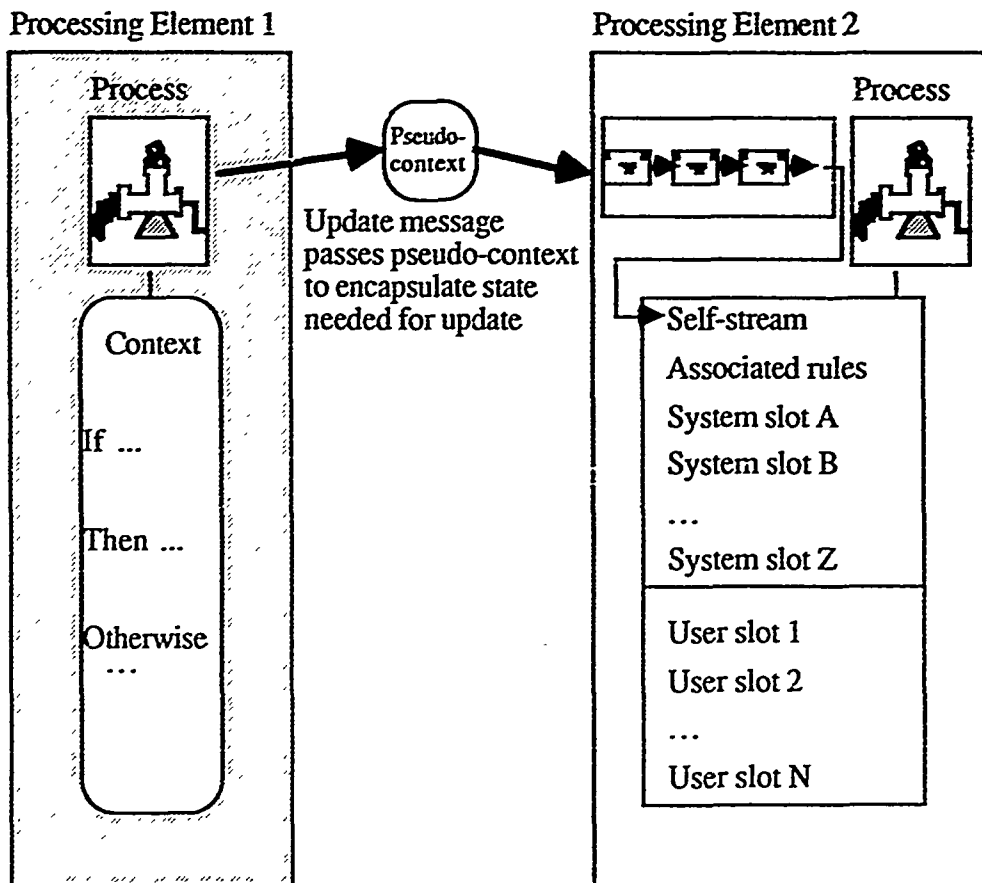


Fig. 4-15. When an update is required, a pseudo-context representing the required state from the current context is passed along with the update message to the node to be updated. The update is performed in the environment of the pseudo-context. Copying the state in the context allows concurrent execution of action parts without contention for the context in which the rule is executed.

When the message for the update arrives at the node to be modified, it executes the necessary code to perform the update, extracting any definitions that it may need from the pseudo-context that it has been passed. As mentioned earlier, the compiler has already determined which definitions will be needed in order to perform the update. At run-time the context that evaluates the rule knows which of these definitions have indeed been evaluated. As long as all the needed definitions have been evaluated there is no problem – the update is made, and nothing more needs to be done. A problem arises, however, if not all of the definitions have been evaluated because a deadlock can occur if some special mechanism is not incorporated. This is described below.

As was already mentioned, the only way Polygon allows the user to express serialization is by the serializing of the action parts of rules. The following code fragment executes the changes requested serially;

Action Part :

Changes :

Change Type : Update

Updated Node : «aircraft 1»

Updated Fields :

wheels ← «a new wheel»

wings ← «left canard», «right canard»

Change Type : Update

Updated Node : «aircraft 2»

Updated Fields :

wheels ← «tail wheel»

wings ← «new left wing»

Here the system must wait until the update to «aircraft 1» has finished to execute the second change. In turn, the context executing the rule must wait for confirmation that the first update has occurred from «aircraft 1» before it can perform the update to «aircraft 2». The context, therefore, waits on the stream from which it expects to get the reply confirmation. This is not a problem unless the pseudo-context that was passed on to perform the original update has not already evaluated the required definitions. If such is the case, the execution of the update will require a fully fledged context in order to execute the action part because the code for the evaluation of the outstanding definitions is, by specification, allowed to block for futures at any point.

Since the design of Poligon requires that code executed on blackboard nodes not be allowed to block, the execution of the update has to punt to another context. It is not possible to ask the original context to evaluate the missing definitions because that context is already blocked, waiting for the reply from the update that is in trouble. This is a deadlock condition.¹ A new context is created to perform the update instead, a relatively expensive process; and although the semantics of a rule that punts in this way are very similar to one that does not punt, the system issues a warning message that this is the case so that the programmer can rewrite the rule to force the evaluation of the missing definition. This process is shown in Figure 4-16.

¹Other mechanisms for deadlock avoidance are possible here, this was just the simplest mechanism given the strongly futures-based implementation model of Poligon.

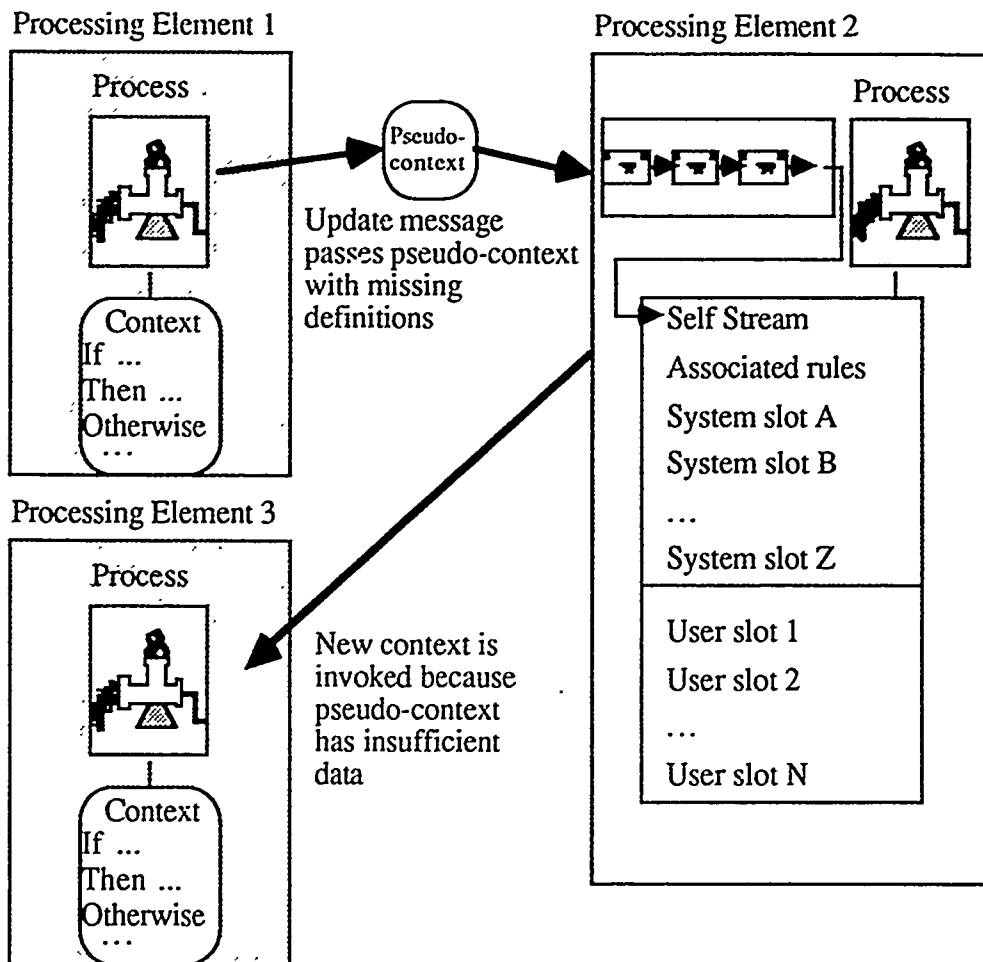


Fig. 4-16. A rule executing in the context on processing element 1 requests an update of a node on processing element 2. The pseudo-context that is passed to the node to be updated does not have enough of its definitions evaluated, so it allocates a new context on processing element 3 to evaluate the missing definitions and to finish the update.

Once the update has been performed either on the node using the pseudo-context or through the agency of a new punted-to context, the original context that is evaluating the rule is sent a message to confirm that this has happened. The context can then synchronize correctly and continue with the evaluation of the serial parts of its rule.¹

4.7.4. Expectations

The farther the experiment is from theory the closer it is to the Nobel Prize.

— Frédéric Joliot-Curie

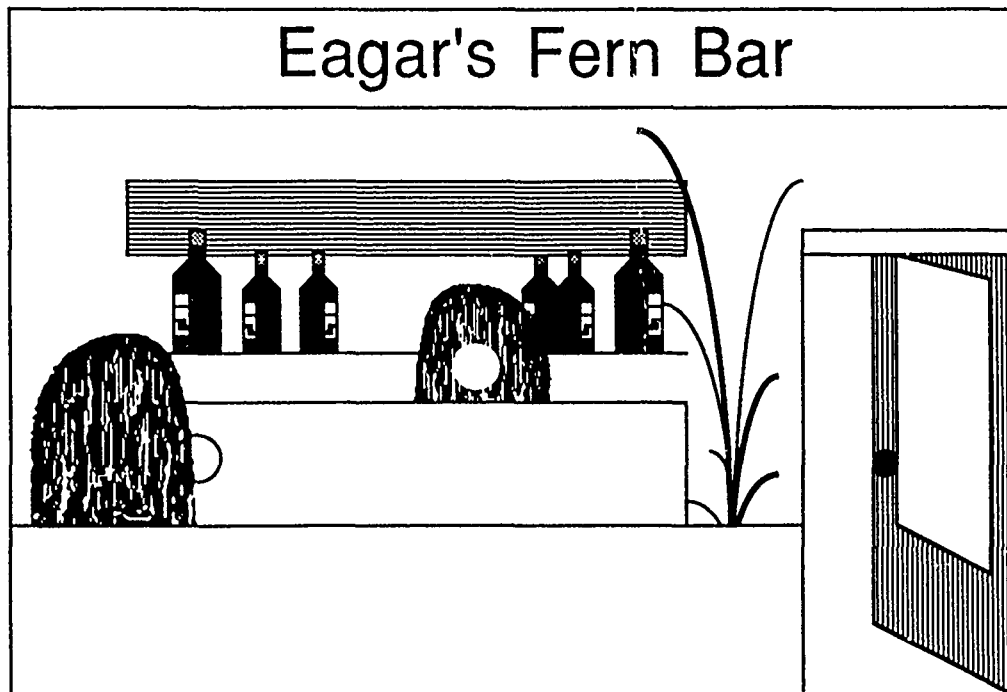
Mention was made in Section 4.7.1 that the slots to which rules are attached are defined at compile-time. The exception to prove this rule is discussed in this section.

¹In retrospect, this design is inadequate, but deciding on a good implementation for these semantics is difficult. A change to the specified semantics of the language is likely to be the best way around these implementation problems.

An expectation mechanism, a common part of blackboard systems, allows the user to express the fact that some particular event is anticipated. This allows model-based reasoning to focus the attention of the program on places of importance. We wanted to have some way to take advantage of such model based reasoning in Poligon. This section describes the implementation of Poligon's expectations.

Poligon's expectations are dynamically allocated rules. Just as normal Poligon rules are associated with slots at compile-time, expectations are associated with a particular slot on a particular node at run-time. Expectations allow the effective focusing of attention on a particular node while still saving the overhead of associating that rule with all nodes of the class involved. Moreover, state in the rule that initiated the expectation may be used to specialize the rule being allocated so that the behavior can be more highly focused. Because a real-time blackboard system is just as likely to be interested in something not happening as something actually happening, a timeout mechanism supported by Poligon's expectations allows them to wake up after a predetermined time, knowing that they have timed out.

Expectations are represented as structure objects. When a rule decides to post an expectation, it sends a message to the node that will be focused on asking it to record the new expectation. When the expectation is formed, the programmer has the option of defining arguments to the expectation rule and of passing in extra conditions for the When and If parts of the rules. These allow context-specific state to be allocated with the expectation. For instance, if an aircraft is expected to land at a specific airfield, one might post an expectation on the airfield that asked the airfield to look out for aircraft landing. One would also pass it the specific aircraft as an argument to compare with arrivals so that it can know it has noticed the aircraft that is interesting to the program.



Expectation.

Extra slots in the expectation data structure allow the user to specify whether the expectation is active or not or the number of times that the expectation is to try to fire. This is important because it may be that an expectation is valid for only a highly focused set of cir-

cumstances. For instance, the aircraft in question can only land after it has had enough time to fly to the airfield. We do not want expectations to fire off all of the time, and, if the particular aircraft does land, we want the expectation to decouple itself from the node it is watching. All of this is possible in Poligon's expectation mechanism.

Similarly, we may want to know if an event has occurred after a certain time, if the aircraft fails to land at the specified airfield after a given period, for example. This would indicate that our model of the aircraft's actions was wrong and we have to reevaluate what is happening. Such an occurrence is also expressed when the user posts the expectation. The timeout that is to be made is encapsulated on the node being watched, and the set of timeouts that are still pending is examined each time the domain real-time clock ticks. It is an attribute of all Poligon nodes that they can know the real time and that they can be sensitive to clock ticks. When the clock ticks past the time specified by one of the pending timeouts, the rule associated with the expectation is fired and its Timeout Part — a component much like an Action or Otherwise Part — is executed, allowing the program to take whatever corrective action is required.

This timeout mechanism does have a problem, however. In order to work reasonably well, either the system must be lightly loaded or the timeout duration must be long with respect to the time scale of events in the system. This is because a Poligon program can cause the processors on which it runs to become highly and unpredictably loaded, resulting in significant delays in computation. Thus, a timeout might trigger simply because the program was being held up, not because an event failed to occur, i.e., the plane failed to land. Clearly, a Poligon program must not be brittle with respect to timeouts triggering in this way.

At this time, Poligon's expectation mechanism has never been used in an application. This could cause one to lose faith in its usefulness, but we still have some hope for the value of this mechanism, since they may indeed be more useful in a genuinely real-time environment. The knowledge that we implemented for our applications was already expressed in a strongly non-model-based manner, and the real-time aspects of the applications were not really concerned with producing responses that the real-time clock might have triggered. For this reason, the timeout mechanism was not useful. Similarly, in order to be able to compare our experimental results with those produced by the Cage [Aiello 88] and Lamina [Delagi 88b] projects, we were compelled to make the application's problem solving behavior much like that of the others. Because of the way in which the use of expectations affected the problem-solving process, we could not easily produce results comparable to those of the other implementations, so use of the expectation mechanism was removed.

4.8. User Code and Definitions

As you will recall, Poligon supports a mechanism that allows the programmer to express ideas much like knowledge source bindings in an AGE program. These are called *definitions*.

The idea behind definitions was to support the functionality of AGE's knowledge source bindings without suffering from their defects. Nevertheless, Poligon's implementation had a set of defects of its own.

We decided early on that definitions should be lazily evaluated. This was due to an æsthetic preference and a desire to around a deficiency in AGE's implementation. In AGE it is common to define a knowledge source binding with a null value at the knowledge source level and then to set *q* in a value for it if the value is needed. This substantially compli-

cates the program and obscures the programmer's intent. The plan for Poligon was to have a means of associating names with values and having the expressions that delivered those values execute once at most, but not at all unless the value was needed.

We implemented definitions by making a mapping from names to structures in the context objects that represent the invocation of rules. Thus, for each defined name there is an entry in an AList that associates with that name a value or a token denoting that the definition has not, as yet, been computed and a function that will compute the value if it is needed. In a production-quality Poligon system one could optimize this implementation significantly, but for our purposes an AList was adequate.

When a user made a reference to a definition, the compiler converted it into an access function on the context object that would compute or unpack the value as appropriate. For example,

```
Definitions : four  $\equiv$  2 + 2
When : four = 4
```

would expand during compilation into something of the form

```
When : (eq1 (get-value-from-definition :four
      _the_current_context_)
      4)
```

where `_the_current_context_` is the name given to the context object that is visible during the evaluation of the When part of the rule. The function `get-value-from-definition` would extract the required value or would compute the value with a function that the compiler generated to represent the expression `2 + 2`.

We quickly found that we had to represent multiple values in definitions, so we used a slightly modified implementation to unpack multiple-valued expressions into their component values.

This implementation was somewhat naïve because it again assumed the existence of an efficient, blackboard machine that would implement this sort of behavior effectively. We discovered, however, that the performance of our applications was being limited by the use of these definition items. The cost of extracting an already computed value from a definition was too high. We could have taken two approaches to address this problem. First, we could have worked to optimize the implementation of definitions, or, second, we could have used a better compiler. Because of the convenience associated with the existing implementation of contexts and their definitions, we decided to try the latter approach.

Our problem derived from code fragments such as the following:

```
Definitions :
    two  $\equiv$  2
    all-wheels  $\equiv$  the-aircraft@wheels
When : the-aircraft.is-airborne and
      all-wheels.length = two + two
```

In this case example the expression `two + two` makes multiple references to the same definition. From the semantics of definitions we know that each reference to the name `two`

will always have the same value. We cannot, however, extract all the definitions in the When-part expression and evaluate them all first because of the short-circuit semantics of operators like And or because of conditional expressions. Not taking notice of these would destroy the lazy semantics of definitions. The compiler was made sensitive to these concerns and transformed the previous expression into something like the following:

```
When :
  (and (get-slot-value the-aircraft :is-airborne)
        (multiple-value-bind (_two_ _all-wheels_)
          (get-multiple-values-from-definitions
            the_current_context_ :two :all-wheels)
          (eq1_(length _all-wheels_)
                (+ _two_ _two_)))))
```

Here, all of the required definition values for any given lexical level – working outward from deep lexical levels until a non-strict operator or condition is reached – get extracted as a block, and the values are seen in local variables introduced by the compiler. This made a substantial improvement in the performance of user code.

The performance of a system such as Poligon is significantly restricted by the copying of the definitions template into the context objects from the rules to be executed. The definition template is represented as a list of lists, so it is implemented as a *copy-tree* in Poligon. To be sure, this is a suboptimal implementation. A better alternative would be a positional representation of definitions. This would allow the template to be represented as an array, and the initial copying of that template could be a simple block transfer operation. A positional implementation of definitions can easily be made because all the definitions are known at compile-time. The extra effort in reimplementing from the inadequate, original design prevented us from trying it in Poligon.

4.9. Search



Search.

Experience with existing serial blackboard systems such as AGE and MXA caused us to believe that real-world blackboard systems are likely to spend a significant amount of time and effort performing searches over the blackboard. The reason is that these systems frequently need to be able to correlate new pieces of data with the existing solution. For instance, if new blips come into the system from radar detectors, being able to associate these with the aircraft that caused them is important. Making that match often involves searching all the aircraft.

A serial search through the blackboard is usually a linear time process at best. Being able to perform the search in parallel should allow this in principle to happen in constant time, ignoring the overheads and problems associated with parallel processing. We decided, therefore, to support this sort of search operation at the Poligon language level and implement it in as efficient a manner as possible.

Search is implemented in two forms in Poligon that could be efficiently implemented, in a real-world system.¹ Searches over blackboards usually consist of matching a value against a slot in a node. Failing this, they consist of matching some arbitrary condition for a given node. In the first case we were able to encapsulate this request for a match in a simple message to all the nodes being searched. The message contained the value to be compared to, the name of the slot to be compared with, and the operator to be used in the comparison. The second, more general implementation required that a closure be formed over the things to be compared and that the closure be evaluated by each of the potentially matching nodes. This is clearly less efficient than the first case but can still be encoded respectably.

Once we had developed a mechanism for deciding which nodes match, we had only to determine which nodes to search and how to process the replies from the search messages. Poligon, because it is implemented on a machine that supports multicast messages, is able to send the message for the search to a large number of nodes efficiently in one multicast message. Poligon allows the user to search over either a collection of nodes or all the nodes in a class. If the user want to search over a class then the context performing the search sends a message to the class node, which then forwards the search request by multicast to all of the nodes in question.

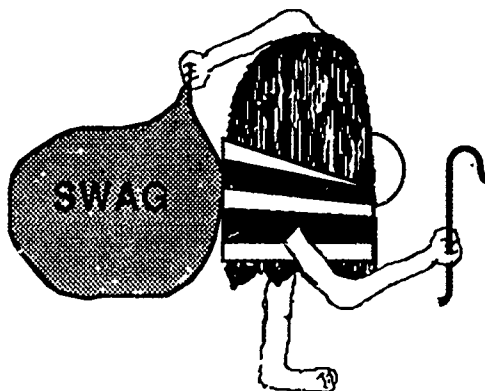
When the nodes being searched reply to the search request, they all must send messages back to the context that initiated the search. It is not possible for only the matching nodes to reply, because the context performing the search will never know whether it has all of the replies. The context object must therefore have a means of handling all these replies, many of which may simply say "not me." Poligon uses *bags* to implement this behavior. These are described in Section 4.10.1.

4.10. Data Types

Poligon supports some data types not commonly found in other systems: bags, futures and multiple-values objects. In this section we discuss the implementation of these data types, the reasons for their existence, and their benefits, if any.

¹This is entirely unlike the search mechanism in MXA, which constructed tuples of nodes that matched a certain condition. It was the assumption behind MXA that search would be the overriding cost in the execution of the system, and therefore its model of parallelism required that the blackboard should reside in an associative memory so that these sets of tuples could be constructed efficiently. Unfortunately, no concurrent implementation of MXA was ever made, though the initial candidate for it was an associative database machine.

4.10.1. Bags



Eagar thinks that bags are useful.

Poligon uses bags to handle replies from searches. Bags have the valuable property that they are *unordered* collections of values. This means we can process the values in a bag in any order we want; the ordering does not matter to the program.

When the user performs a search operation such as that specified by the expression

Subset of Aircraft Such that Element • wings = 2

the value returned immediately is a bag that represents the matching values. This bag is implemented as an object that contains a list of values that have already been determined and a means of generating the values that have not yet been determined. In this case the means of determining the other values is a data type called a *multifuture*. This is a data structure connected to the stream to which all search replies will be sent. As the user tries to extract values from a bag, it returns values from the determined elements first, and if this is empty, it looks on the stream for any new values, only blocking the searcher if there no values are there. If a value found on the stream is a "not me" reply, this is discarded and the bag tries again to get a useful value. The result of this design is that the process looking for the values in the bag will always have access to any new values as soon as they arrive — it doesn't have to wait for a particular element to be returned — and will block only if no values are present.

This implementation gives a clean and abstract interface to the searching process and seems to work very well. Its major deficiencies are that the network can get congested with all the replies to the messages (see Figure 4-17) and that a regular mechanism for implementing collection data types is difficult to optimize without specialized hardware or microcode support.

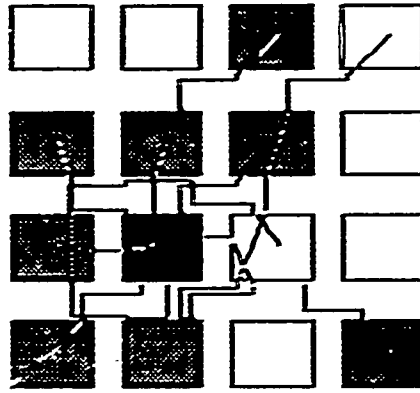


Fig. 4-17. An instrument from the CARE simulator shows the network around a processing element becoming overloaded with a large number of multicast replies.

4.10.2. Multiple Values

Common Lisp supports the return of multiple values from functions. Many implementors of Common Lisp believe that the purpose of multiple values is to avoid CONSing, that is, to save the programmer from having to CONS up a list that denotes the values to be returned and then expecting the caller to unpack the list that was returned.

According to another school of thought, the purpose of multiple values in a language is to express the fact that many functions logically should return more than one value. It is not necessarily meaningful to define a new data structure type for each function's returned values. All the values of a function call are meaningful, however, and there could well be purpose in preserving them. The point here is that in the Common Lisp case, multiple values are CONSed onto the stack and are discarded as soon as possible. In the other case — and Polygon is an example of such a system — multiple values are CONSed into the heap and are discarded as late as possible, i.e., when they encounter a strict operator.

The existence of persistent multiple values is clearly motivated primarily by linguistic aesthetics. Nevertheless, the marginal implementation cost of introducing multiple value objects was small, given that the system already had to be sensitive to strict operators in order to support its model of futures. Multiple values were implemented simply as named structures with a slot containing the list of values. A production-quality system would presumably have a more appropriate implementation. Although somewhat prone to CONSing, this implementation of multiple values seemed to work well, was fairly simple to use, and removed any ambiguity that the transmission of multiple values between processing elements by the system might have caused. There are enough entry points from the user's application into the underlying Polygon implementation that it would have been difficult to preserve the semantics of the native Common Lisp's multiple values implementation when they were viewed from a Polygon application.

4.10.3. Futures

Unlike existing implementations of systems with futures on real hardware, as opposed to Polygon's simulated machine, Polygon was unable to have low-level support for its implementation of futures. A number of ways in which the Polygon implementation dealt with this issue have already been mentioned. In this section we discuss the implementation of futures themselves.

Polygon supports two forms of future, futures and multifutures. Neither are accessible at the Polygon language level, though when data structures are printed out they often appear in

a form somewhat like (1 2 3 #<Future 4> 5 #<Future Unsatisfied>), where #<Future 4> is a future that has been satisfied and has the value 4 and #<Future Unsatisfied> is a future whose value has not been computed yet or whose value has not yet reached the owner of the future.

Futures in Lamina are not first-class citizens. They are simply specializations of streams that return only one value. They cannot be trivially passed around between objects on different processors. In Polygon, considerable effort was spent on trying to make the implementation of futures as seamless as possible. Futures are named structures that point to the streams that deliver their values. A flag is used to indicate cheaply whether the future has been satisfied or not. Streams live on the particular processing elements on which they were created. As a result, special support has to be provided to allow futures to be passed around between processors.

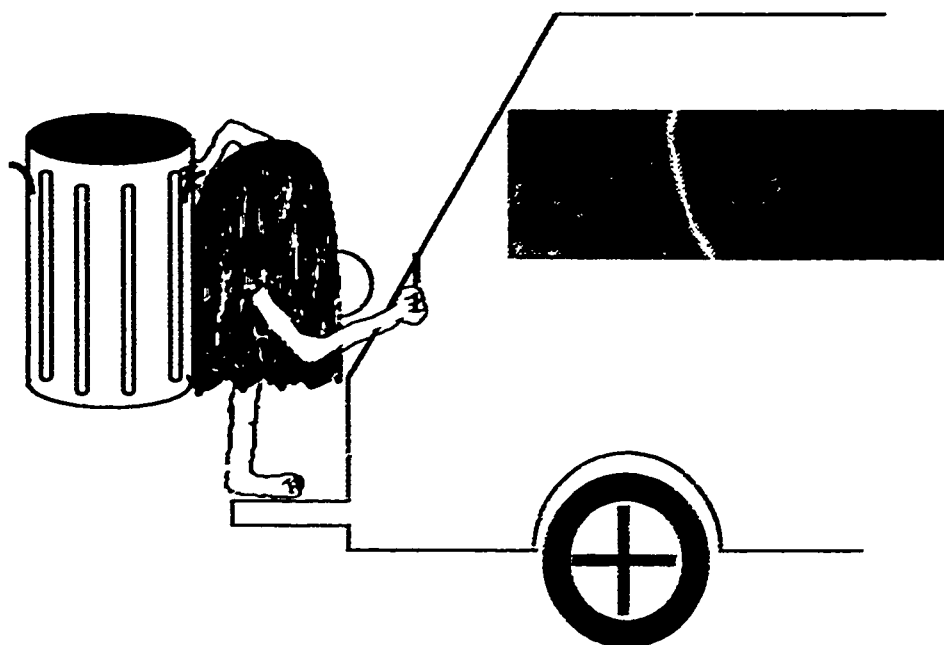
When an unsatisfied future is passed to another processing element, a remote address to the stream of the originating future is passed along with the future. The copied future therefore has a back pointer to the old stream, but nothing more is done. The future is modified on the originating site so that the future points to a new stream. This stream is then linked to the original stream in the future. It is a property of streams in the CARE machine model that they can be linked, so that values that appear on one stream will be forwarded automatically to any streams linked to it. Thus, if the future that was copied to a different site is ever defutured, it forms a link to the old stream and then waits for any values in the stream to be passed to it. This is a sort of forwarding pointer implementation using message passing across the boundaries of processing elements.

Multifutures implement the bags used in blackboard searching (see Section 4.10.1). They are much like futures in that they are named structures that point to streams. The main difference being that multifutures generally receive more than one message and futures only receive one reply. Multifutures could present a problem to the system in the general case because they do not know how many values they are expecting. They would therefore not know when to relinquish the resources they were using and allow themselves to be garbage collected. In the case of Polygon's use of multifutures, however, the bag that creates the multifuture always knows how many values it is expecting. When the right number of values have returned, the bag can drop the multifuture.

4.11. Optimization

The original implementation, although intended to be highly compilable, was woefully inefficient. This was simply because we were more interested in investigating the concurrent problem-solving process than in making hard measurements of the resulting system's performance. Eventually, however, we had to try to improve the performance of the system in order to get reasonable results from our experiments. These experiments are documented in some detail in [Nii 88a] and [Rice 88b]. In this section we discuss some of the optimizations we introduced in order to improve Polygon's performance. We developed many of these optimizations to provide efficient support for special cases of generic operations. Consequently, many would not have been necessary if Polygon had not provided as general and abstract a model to the user.

4.11.1. Collections



Collection.

Poligon supports a number of different collection data types, for instance lists, bags, and sets. Because the original implementation of Poligon assumed the existence of specialized hardware to deal with the data types that we wanted to introduce, to implementing a powerful set of generic operations for all collection data types seemed like a good idea. Bags are implemented as Flavors instances, as are sets; lists are just lists. We noticed significant performance degradation in applications from the use of the generic operations that Poligon supports in order to manipulate these data structures. For instance, `foo-head` will extract an element from a bag or the first element from a list. The need to perform numerous typecases on everything meant that the application was unable to take advantage of the efficient, microcoded support for list operations. Likewise, because the Poligon system knew little about types, the compiler tended to introduce far more defuturing coercion operators than was strictly necessary. As a result, for almost every argument to every function a Poligon system coercion function was called. This introduced a significant performance penalty.

To alleviate these problems, we implemented considerable support for the declaration, inference, and propagation of types into the compiler. Because the subset of the language in which most of a Poligon program is written is side-effect free, we were able to take advantage of the fact that the type of the value associated with any given identifier does not change within the scope of that identifier. This means that the compiler was able to propagate inferred and declared types much more effectively.¹

As an example of this, consider the following expression:

¹It should be noted that this was only possible because we had access to the Lisp compiler's source code, since Common Lisp did not specify any user accessible interface to the definition of compiler optimizations or to type information. We therefore would like to express our gratitude to Texas Instruments Corp. for its excellent source-code distribution policy.


```

Let result ≡ argument • tail In
  Map-Over-A-Collection( #'÷, result, 2)
EndLet

```

In the original Poligon implementation this would have compiled into the following Lisp code:

```

(let ((result (tail (↑ argument))))
  (map-over-a-collection #'÷ (↑ result) 2))

```

In the above code the function \uparrow is the defuturing operator. The `map-over-a-collection` function maps its first argument over the collection denoted by its second argument. As each element is extracted from the collection, it will be put through the \uparrow operator in order to apply the \div function to it. In the end, a collection that is the same shape as the original collection will be returned. The value of the `result` identifier will be a collection of the same type as that denoted by `argument`, only missing an element.

After the inclusion of the type-checking code we were able to do the following:

```

Has-Type(argument, list, number)◊
Let result ≡ argument • tail
In Map-Over-A-Collection( #'÷, result, 2)
EndLet

```

Here the type declaration declares that `argument` is a list of integers. Knowing that the `tail` function is being applied to a list allows the compiler to deduce that `result` must also be a list of numbers. Knowing that `result` is a list of numbers allows the compiler to open code the mapping operation. Similarly, the compiler knows that each element of the list over which it is mapping is a number, so it knows that it will not have to apply any defuturing coercions to the elements of the list during the evaluation of the code. This generates the following Lisp code.

```

(let ((result (rest argument)))
  (loop for .element. in result
        collect (+ .element. 2)))

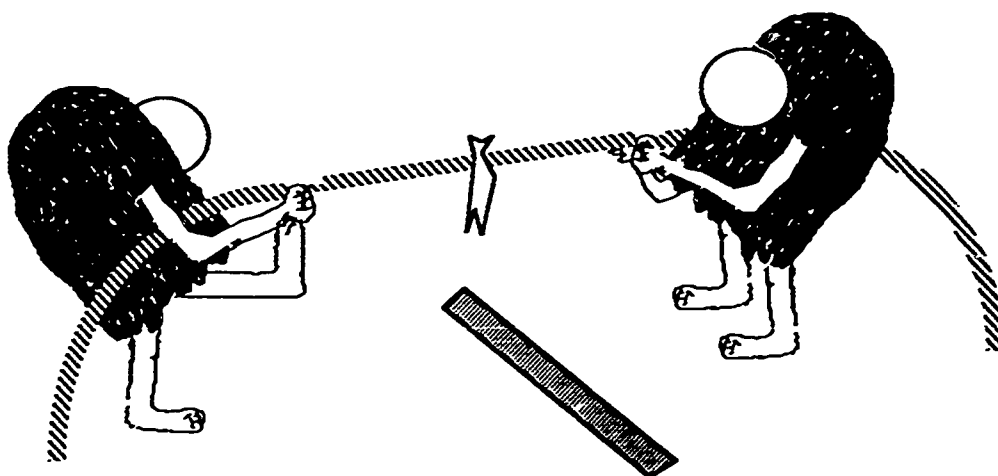
```

Such code will compile into just a few instructions rather than a large and complex set of function calls. The speed improvement in such cases is easily of the order of 20x.

One problem we observed was that the use of such extensive compiler optimization made code harder to debug. Because of the way the compiler open codes Poligon's collection-processing functions, a few lines of source code can often expand into tens of lines of Lisp code, a complication when one lands in the debugger. To address this problem we made the compiler optimizers in Poligon sensitive to the `Optimize` switches in the underlying Lisp system. If the program is compiled for low speed and high safety, then the user gets everything from run-time type checking of named structure accesses and Lisp-level source code debugging. If the code is compiled for high speed and low safety, then run-time type checks are compiled out, structure accesses compile into `arefs`, and collection-processing operations get open coded whenever possible. This use of the `Optimize` switches proved worthwhile and was used for all Poligon's optimizations.

Clearly, the code shown above is no better than what would have been achieved had the user written everything by hand. Yet, this approach seems useful in practice because the user often does not know which data structures will or will not contain futures or the like, being able to make only occasional assertions about the implementation types of various expressions. This approach seems to decouple the user effectively from the implementation of his data structures and still allows improvements in performance by the addition of type declarations, which do not affect the semantics of the program. This means that optimizations can be achieved without having to rewrite any code.

4.11.2. Equality



Equality.

Poligon needs to have its own idea of equality: it must be able to compare data structures that may have been copied from arbitrary places and get the right answer. Similarly it needs to accommodate special handling for the comparison of futures. It is very undesirable to block on the comparison of futures unless it is strictly necessary. Poligon also needs to have some reasonable behavior to allow the comparison of multiple value objects.

It is for these reasons that the generic Poligon equality-testing operator is very complex and considerably more expensive than the microcoded equality-testing predicates supported on the native machine. The Poligon system was wasting a significant amount of time in making expensive comparisons, so the equality-testing predicate was an early target for compiler optimization. In most cases the use of type declarations and type inference allowed us to compile uses of the = operator into calls to the microcoded eq and eql operations. Because of this, the examples of generated code in this paper show, references to the = operator as being compiled into calls to the appropriate Lisp predicate.

4.11.3. Slot Reads

Just as type declarations are used to optimize the manipulation of collection data structures, so they are also used to optimize accesses to slots. Poligon originally had a slot read implementation that sent far too many messages. A message was sent to the node asking for the slots to be read, each of these would result in more messages being sent to extract each of the slots involved and then still more messages to get the required values from the slots themselves. This need not be necessary, especially if you know that you are executing code in the process associated with the node, which is always the case for the slot evaluation functions mentioned in Section 4.4.3.

Our approach was to use the type declaration mechanism both to declare the types of the nodes being manipulated, and to declare our knowledge that a given function was to be executed directly by a node. We were, therefore, able to write code such as the following:

```
Define some-function (arg1, node)
  Has-Type(arg1, list, number)()
  Has-Type(node, aircraft)()
  node.wings.length = arg1
EndDefine
```

Now because we know where this code is to be executed, we can compile it into something like the following:¹

```
(defun some-function (arg1, node)
  (declare (self-flavor aircraft))
  (eq1 (first (send wings :values)) arg1))
```

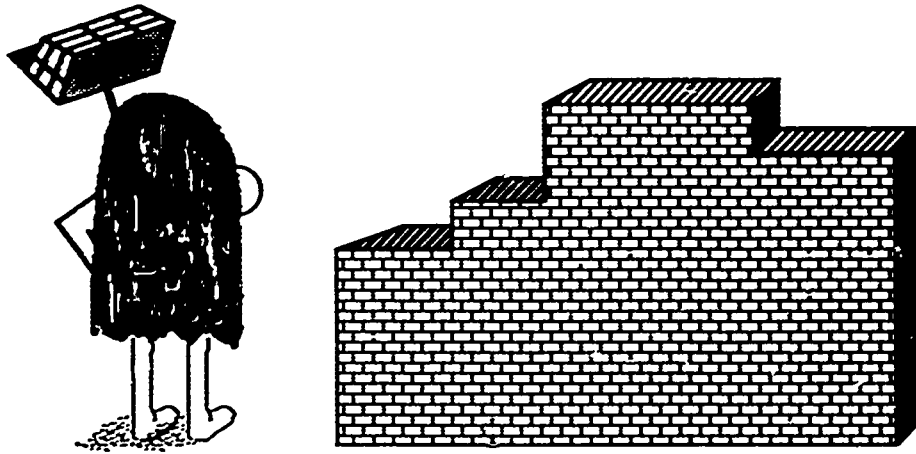
Here the message `:values` is being sent to the slot object denoted by the slot name `wings` to extract the values associated with it. Such code would not be necessary in a production-quality system that did not implement its slots as objects being pointed to from its nodes.

4.11.4. Block Compilation

Block compilation is one aspect of program compilation and optimization that Lisp implementations generally avoid. We did not want to be limited in this way but clearly had to accept that we had no reasonable way of block compiling our Lisp code. We knew, however, that we had the option of block compiling our knowledge base. We found that a significant amount of time was being spent in knowledge search, and this encouraged us to investigate the block compilation of our knowledge base.

In a system such as OPS [Forgy 76], most of the system's time is spent in performing a search over the knowledge base for applicable rules. Blackboard systems, by the very nature of the way in which they are decomposed, have had a substantial amount of knowledge search hand-compiled out. The preconditions on knowledge sources allow the rapid filtering of knowledge and the selection of knowledge sources that are interested in particular classes of events. This same sort of hand compilation applies to Poligon because the user associates rules with particular slots in certain classes of nodes. But, when a slot is updated in Poligon, the system must still search the (admittedly small) list of associated rules to see whether any rules are interested in the update. This search is not computationally trivial but can be performed at compile-time. Thus, if block compilation is allowed, the search is not necessary.

¹The actual implementation would be more complex than this, but the example shows essentially what happens.



Block Compilation.

Rules are not associated with the majority of slots in a normal Polygon program. Likewise, most slots that do in fact have associated rules have only a small number, and this number does not change during the execution of the program.¹ Thus, in most cases, it is possible to determine at compile-time all the rules that are interested in all slots. This requires block compilation, since there are always forward references in real code. After the entire knowledge base has been loaded, it is possible to recompile the system so that all slot updates to slots that have no associated rules are open coded in a very simple manner. Slot updates to slots with associated rules are open coded in a manner that wires them directly to the relevant rule objects, thus totally eliminating knowledge search.

This strategy has its costs, however. Once a Polygon program has been block compiled, it is not possible to add or remove a rule without completely recompiling the application. Clearly this sort of operation would only be performed once an application was well debugged, but it is a small price to pay for improved performance. Moreover, this strategy conforms to Polygon's philosophy, which is to trade extra compilation time for improved run-time performance.

4.12. Signal Data Input

On trapping a lion in a desert [Petard 38]: The Cauchy, or Functiontheoretical, method. *We consider an analytic lion-valued function $f(z)$. Let ζ be the cage. Consider the integral*

$$\frac{1}{2\pi i} \int_C \frac{f(z)}{z - \zeta} dz$$

where C is the boundary of the desert; its value is $f(\zeta)$, i.e., a lion in the cage.²

Because Polygon is a system that at least attempts to perform real-time operations, we needed a simple mechanism for introducing data into the system. We of course lacked ac-

¹Expectations are an exception to this rule.

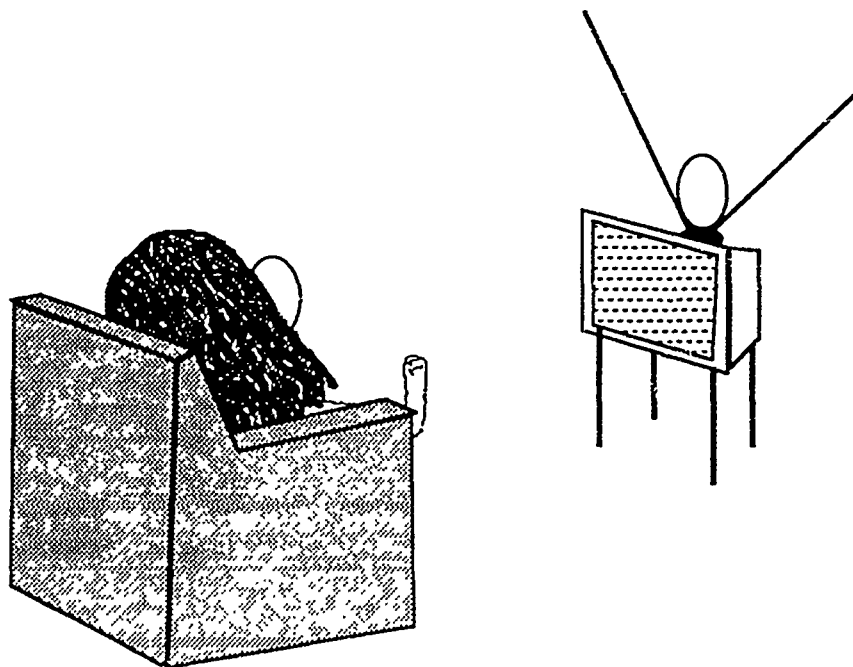
²N.B. by Picard's Theorem [Osgood 28], we can catch every lion with at most one exception.

tual real-time monitoring equipment, but we wanted to get as reasonable an implementation as possible.

All signal data in Poligon gets into the system from one (Lisp) stream. The data in that stream is timestamped and coded so that the application can find out when the event that it denotes was supposed to have happened and what sort of event it actually represents. This stream is read by the class node of a class of input handlers. The timestamp of the node is read and the node sends a message to itself, which is timed so that the node will wake up to process the signal data at the appropriate simulated domain time. When the class node wakes itself up to deal with the signal data, it allocates to itself an instance of the class that it represents to handle the input from a resource kept in one of its slots. It then sends a message to an instance of itself that tells it to process the input. When an input handler has finished its processing, it sends back a message to the class node on a private stream telling the class node that the input handler is free and can be put back in the resource. If there are not enough instances in the resource to handle the signal data at any time the class node creates new instances and sends them initialization messages that tell them to process the input. In practice, we found that the number of input-handler nodes created was generally the same as the number of signal records read in a given timeslice.

Once the input data arrives at the input-handler server node, a user-defined procedure is invoked in order to process the input. This procedure would typically instantiate a node to represent the input that it had received.

In retrospect, we can see that this implementation had a number of deficiencies. First of all, for linguistic reasons, the only thing these input handlers could really do is create nodes; the user could not update an existing node, for instance. It was effectively always necessary to represent the new data as a node before the system could do anything about it. This is a deficiency because it tends to create a large number of nodes that are not necessarily useful to the representation of the application. In addition, if the programmer wishes to view the process of data arriving in the system as a message-passing process, Poligon will not allow this model.



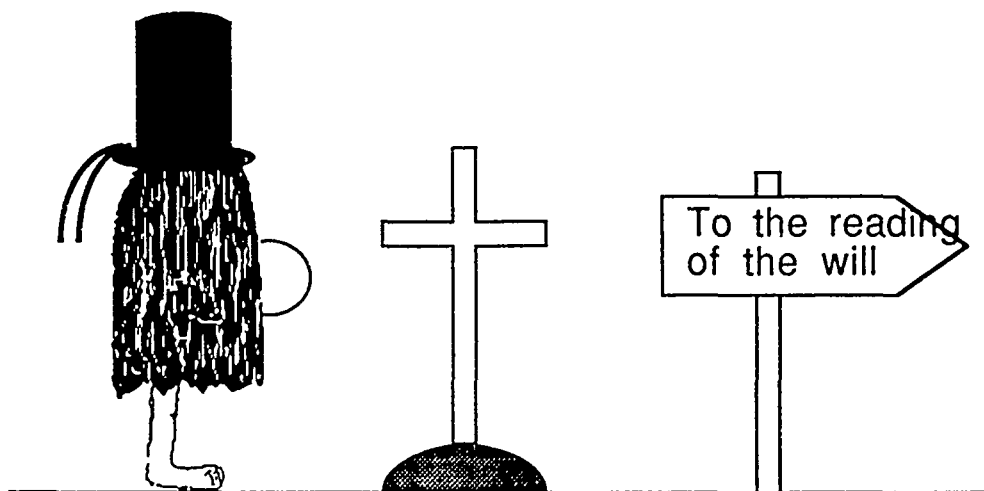
Signal data input.

The second major deficiency of this design is not so much structural as one of execution. It seems worthwhile to create new input handler nodes to deal with new signal data if their resource is empty; but if the system gets heavily congested for whatever reason, the input handler creation process can run away, creating masses of server nodes. In a sense this is an artificial state of affairs, because if the program cannot keep up reasonably well with the real world something is wrong with the program. It is, after all, supposed to be a real-time program. Nevertheless, some limit to the number of input handlers is likely to be needed in order to stop the act of loading data into the system from overwhelming the system. How to compute this limit is a problem we have not addressed. In most of our experiments we were intentionally running the system in a manner that would not overload it excessively. This meant that the system's performance was not adversely affected by input-handling activities. It seems likely, in the absence of any analytic model for Poligon's behavior under such conditions, we would have to perform extensive experimentation in order to find heuristics that would allow us to limit the number of input handlers effectively. This is clearly not a trivial problem since the value would certainly vary with different numbers of processors and varying system load.

4.13. Problem Areas

Early work with Poligon yielded the implementation of considerable functionality, whose ultimate utility was unknown. This is not entirely surprising, given the experimental nature of the project. As Poligon developed, some aspects of the system's behavior gained significant importance; others had to be modified in order to make them useful. Some aspects of the system remained unused and were eventually removed, either because these facilities were never used in our applications and software rot set in or because the initial, naïve implementation was not reasonable in the context of later implementations and our developing understanding of the issues involved. This section concentrates on the aspects of Poligon that did not work as planned.

4.13.1. Property Inheritance and Links



Property inheritance.

A number of existing systems support property inheritance and/or some sort of link mechanism. AGE had a somewhat primitive link mechanism. BB1 supports links along which property inheritance can occur. Many systems also support a limited set of system-defined

relationships. For instance, KEE^{TM1} supports the *instance-of* and the *subclass-of* relationships, as do AGE and BBI, though to a somewhat lesser extent. In developing Poligon, we knew that these relationships would come for free from the implementation. They did not seem sufficient, however, particularly because they do not allow the representation of the *part-of* relationship. Any system like Poligon has a problem with the efficient implementation of relationships. A fixed number of relationships can easily be wired into the system, but any user-defined relationships are likely to be harder to implement and less efficient.

The initial implementation of Poligon came with the *instance-of*, *part-of*, and *subclass-of* relationships built in, and we suspected that we needed something more than this. The initial implementation allowed the inheritance of properties along the *part-of* relationship. Thus, if the program attempted to read a slot on a node that was not present there, the node of which the node in question was a part would be asked for that slot, and so forth up the hierarchy. As this approach appeared insufficient, we proceeded to implement a fairly generalized link mechanism along which property inheritance could also occur. Links were represented as nodes themselves, for reasons of regularity, and system-defined slots on each node would contain a list of these links encapsulated within structures that specified the names of the links. System functions allowed the user to find the nodes linked to a given node by a given relationship.

The links were implemented in such a way that, as property inheritance occurred along a link, a system-defined slot would be triggered so that the user could add rules that were sensitive to the act of property inheritance. This implementation appeared regular, but was very expensive. It also had a number of other deficiencies.

- The implementation of user-defined links was not the same as that of system-defined relationships, so the same mechanisms could not be used to manipulate these different links in a reasonable way.
- Deciding on the semantics of property inheritance, although the algorithm for it was well defined, caused considerable problems. Because we did not know what behavior was appropriate for this sort of inheritance, the implemented behavior was probably not sensible.
- The inheritance algorithm specified that inheritance would be sought first up the *part-of* links and, failing that, along any user-defined links. The *part-of* relationship in Poligon is under user control, though reasonable default actions occur in setting up this relationship. This means that a node can be directly *part-of* more than one node. It also means that the set of nodes to be searched for inheritance can be circular and that a considerable amount of effort must be expended to avoid being caught in circularities.
- The implementation of inheritance is incompatible with slot access optimization. Unless the user is constrained to state the class of object to be inherited from, slot accesses cannot be optimized.
- With Poligon's current model of stack-group use, the behavior of this inheritance scheme is not implementable in the general case. This is because if a node is asked for a slot that it does not have, it will have to block and ask for the value from else-

¹KEE is a trademark of Intellicorp.

where. Since blocking is not allowed by Polygon nodes, this design cannot be used.¹

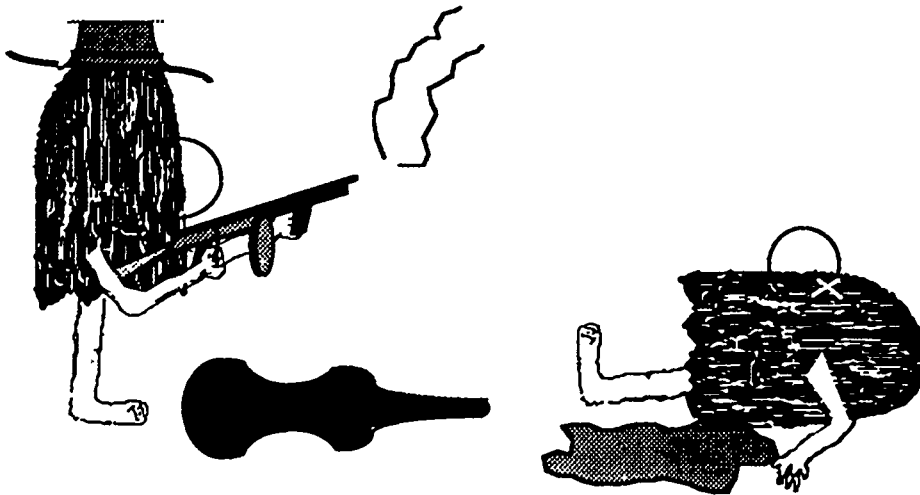
In practice, we found that the property inheritance features were not used even when they worked. This was probably due to a number of factors: the applications appeared not to need it, the abstractions for property inheritance were probably not right, and the inappropriate semantics probably caused users to lose confidence in getting the desired behavior.

4.13.2. Deletion

The deletion of blackboard nodes seems to be a problem that no blackboard system has really tackled. There is an incompatibility between wanting to have a system that knows all about itself and garbage collection. As a result, the deletion of solution-space elements is usually left to the user. Unfortunately, user-defined deletion and resourcing in a concurrent system is an extremely difficult problem. Conventional models for the deletion of objects rely implicitly on being able to determine that either there are no outstanding references to a node or, more commonly, although there are still outstanding references to a node, none of them will ever be used again. In principle such a state of affairs allows the programmer to recycle nodes, but in Polygon this is not necessarily the case. Although the user may think that no more references are going to be made to a node, it is not possible to determine whether there are any nodes still outstanding for that node backed up somewhere in the network. The node might still receive messages that in some way assume it is still the same node even after it has been recycled.

Polygon implemented two forms of deletion for the user: *discarding* and *recycling*. Discarding switched off the node so that no more rules would fire on it but left it able to process any outstanding slot read or write messages. Recycling would completely reinitialize the node and add it to a free list of nodes on the class. The self-stream of the recycled node was replaced with a new one, and the old self-stream was redirected to a site manager object, so that any outstanding messages sent to such a node would be handled in some way — usually by signaling an error.

¹One can envisage a different design in which the node that does not have the slot returns a future to the value of the slot that it doesn't have and sends a message to the node to inherit from, telling that node to reply to the forwarding stream of the future. This works to some extent but prevents the defined semantics of Polygon's slot operations from operating. A multiple slot read or write, for instance, is defined to be atomic. We have no satisfactory way of synchronizing the two nodes in order to get reasonable behavior so the implementation fails.



Deletion.

Although this implementation worked and is still present in Poligon, it was not, in fact, particularly useful for a number of reasons:

- In order for a node to be recyclable, it was necessary to be sure that the reference count to the node was zero. Generally this was possible only if there was just one rule that could fire on the node and if the node would fire that node only once. For this to occur, the node generally had to be at the bottom of the blackboard, created as a result of signal data input.
- As was discussed in Section 4.12, it is not obvious that nodes should be created to represent signal data in the first place. But if they are, the optimized, unmanaged form of node creation should generally be used. Otherwise, large amounts of signal data typically cause the class nodes for the classes created by the signal input procedure to become very hot while servicing all the creation messages. Because the creation of these nodes is usually not managed, it does not have access to the free list of nodes and cannot take advantage of the recycled nodes.
- Discarding nodes generally didn't seem to be useful. It is possible to envisage an application in which the ability to switch nodes off would be useful, but in our applications this did not prove to be the case.

On balance, resource management of blackboard nodes in Poligon is not handled in a useful manner. This is an extremely difficult problem, and possibly the only way to solve it would be to rely on the underlying system's garbage collector.¹

¹Garbage collection, incidentally, is a major area that the Advanced Architectures Project has not investigated. We know it to be a difficult problem. Poligon's use of the CARE model improves matters for garbage collection in some ways, since the only objects that are ever transmitted across a processor boundary are remote addresses or copied data structures with no pointers back to the originating address space. Thus, a garbage collector can always collect any data types other than remote addresses locally. The garbage collection of remote-addresses, however, still remains a major problem. By means of a reference counting model, it seems possible that one could use the CARE processor's communications processor to maintain reference counts as it transmitted remote addresses, but we have not investigated this.

4.13.3. Messages and Events

Messages were eliminated explicitly from the Polygon language. We did not see any particular use for them since they were an artifact of the implementation, and not part of the blackboard metaphor we were trying to represent. This may well have been an error.

Polygon supports a type of action part in rules called *cause events*. The *cause events* mechanism triggers any rules associated with a slot without actually changing any values in that slot. This mechanism was implemented so that the user could trigger rules without having to perform fake updates to slots, which might have caused errors.¹

In practice, the *cause events* mechanism was used as a sort of semaphoring idiom and slot updates were often thought of as messages. Consequently, the programmer had to use Polygon's rule mechanism in order to fake messages and methods. It is possible that the programmers were not thinking in a manner appropriate to the blackboard metaphor, but it is equally likely that the Polygon language lacked generality and flexibility in this regard. If we were to try this again we would certainly attempt to find a better abstract model for the integration of message passing, rule invocation and process management.

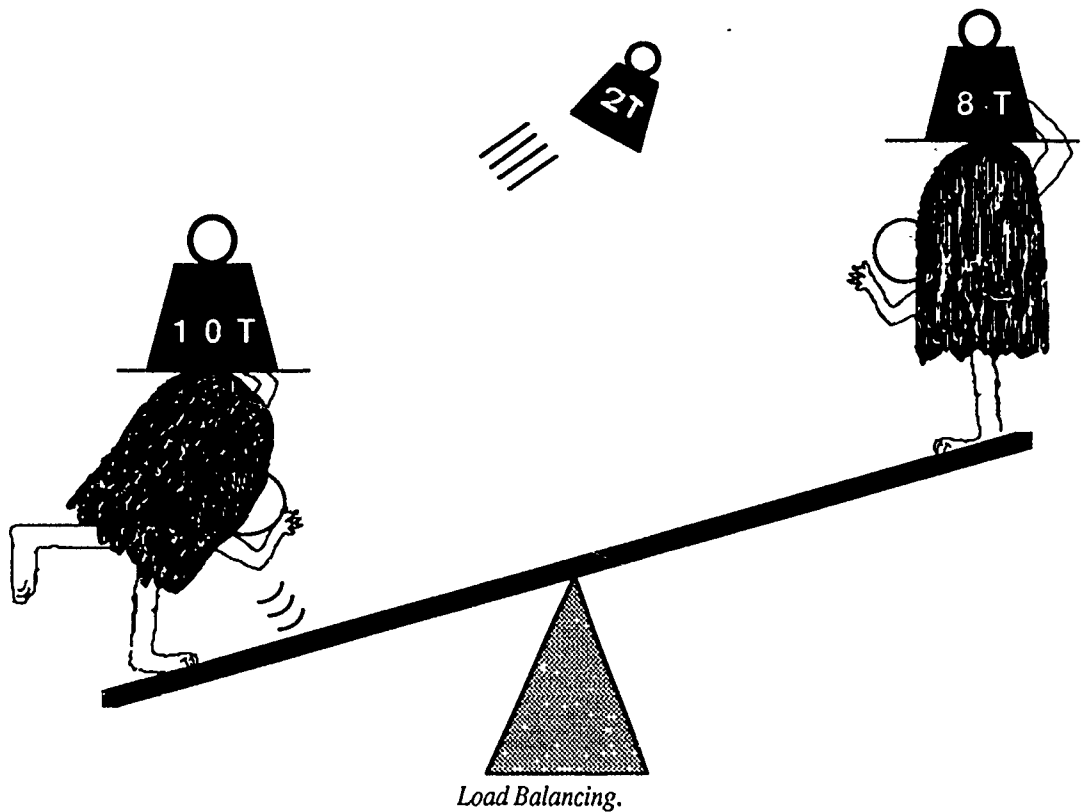
4.13.4. Load Balancing

On the Advanced Architectures Project, load balancing was originally intended to be managed by a layer of software implemented at a lower level of abstraction than the problem-solving system level. Because of the scale of the project, this issue was not tackled until recently and the work on load balancing did not reach any state of maturity until after all of the experiments with Polygon were over. Thus, all the work on Polygon was based on the assumption that a layer of system software that did not exist would exist at some future time.

Our experiments showed that load balance is not a trivial issue. We had originally assumed that we could buy back any performance loss from poor load balance by using more processors and thereby lose only efficiency. This proved not to be the case as was shown admirably in the Lamina Elint experiments [Delagi 88b] and also by the Parable experiments [Bandini 89]. The loss in performance from load imbalance proved not only to be substantial but also unrecoverable. Thus, even though we assumed that Silicon would be cheap at the beginning of the project, we found that this was not enough.

By design, Polygon did not provide any means for the user to know on which processors a program might be running. It was reasoned that, because of the large number of processors, the problem of load distribution would be sufficiently complex that the machine should be able to out-perform the user.

¹Polygon also supports a means by which the programmer can explicitly state that the rules associated with a particular slot are *not* to be triggered as the result of a specific update.



In practice, we found that the user probably should have been given some control over load distribution. For example, the ability to declare that certain classes were likely to create a lot of busy nodes, or that certain class nodes were likely to be very busy.

Ideally, this would be tackled by the environment in some way. It is not difficult to envisage a system that watches the load behavior of a Poligon program and then learns some useful load-distribution heuristics. User declaration of this type would be a second best. Yet, even this model would probably not be sufficient in a fielded system because the problem-solving behavior of the system is so predominantly data dependent. Different behavior in the system will cause wildly differing load characteristics, and so the system would probably need some dynamic load balancing and/or object migration mechanism.

4.13.5. Closures

Closures are one aspect of Poligon that proved to be unnecessarily expensive. This was due to bugs in both the Symbolics and the TI implementations of Common Lisp closures that occurred in the compiling of complex forms such as Poligon application source files. These problems were sufficiently severe that in order to continue with our work we had little choice but to implement our own form of closures. This was not too hard to do because of the semantics of the Poligon language and the existence of the compiler, but the resulting closures, which were implemented as objects, were far less efficient than a native implementation would have been. Systems like Poligon create a large number of closures. An efficient and bug-free implementation of these is crucial to efficient programming.

4.13.6. Pipelines

Our early work on Poligon lacked of understanding of the mechanisms by which parallelism is achieved. As a result, we substantially underestimated the importance of pipeline

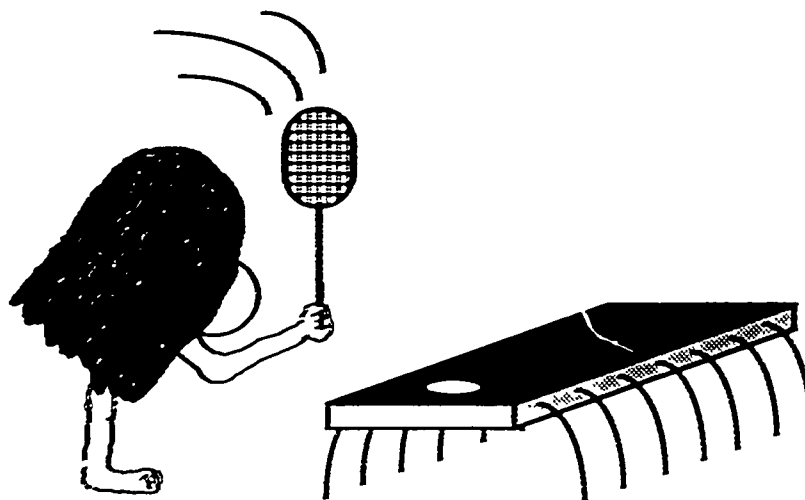
parallelism. In Poligon, pipelines are formed implicitly as data migrates up the abstraction hierarchy, and this may not be the most efficient use of resources. Since the objects in the system communicate with one another by streams, a programming model that encourages the use of non-ephemeral pipelines may be better. This would help to compensate for the cost of stream creation. Lamina [Delagi 86] is just such a programming model. How one could integrate such a programming model with the blackboard metaphor is not at all clear, however. This may be an area in which the underlying system could set up streams between objects and manage them without the user's program having to know about it. We were unable to investigate this area.

4.13.7. Implications for CLOS

The Poligon system was developed using the Flavors object-oriented system before the development of the Common Lisp Object System (CLOS). Although we were generally dedicated to the use of portable standards on the Advanced Architectures Project, we were unable to use them both because they did not exist at the time and also because they would still have been insufficient to give us the level of environmental integration that we sought in Poligon. But, independent of these problems of standardization, there is a more fundamental problem with the new CLOS standard that may not be obvious to the casual reader, but which is likely to be of significance as people start to develop new concurrent problem-solving systems using the evolving standards. CLOS is *unselfish*, that is the concept of *self* has no particular significance in CLOS, unlike Flavors. The behavior of methods is considered to be more closely associated with generic functions than with objects. This has the benefit of giving a regular view of the world and allows multimethods, methods that are specialized on more than one argument.

There is, however, an additional problem with this unifying model; it assumes a shared address space. It is much harder to implement multimethods when the different objects that are being referenced within a method might well be residing in different address spaces on different processors. The implementors of distributed-memory machines tend to think in terms of message passing as the model for communication between both processors and user code. To try to overlay a generic function model of object orientedness on top of this is not a simple matter. We have not had to address this issue because of the immaturity of CLOS, but others in the future will have to think long and hard before they implement a concurrent, object-oriented problem-solving system using CLOS. Certainly, simplifying assumptions can be made. For instance, one could restrict the program only to specialize methods on one argument or only to invoke multimethods on objects that are on the same processing element. Yet each of these simply seems to lead to further complications or loss of generality.

5. Debugging Poligon Programs



Debugging.

Our original motivation in producing Poligon was not just to build a concurrent blackboard system, but rather to build a concurrent blackboard system development tool. Before Poligon was started, a considerable amount of effort had already been expended on the CAOS project [Brown 86] and [Schoen 86]. What we originally assumed would require only a couple of months ended up taking over a year and a half. This was partially due to the immaturity of the CARE simulator, but the difficulty of programming concurrent systems was certainly a major factor. Just as early computer developers would have been hard pressed to envisage a window-based debugging and inspection tool, our first attempts at building concurrent problem-solving systems required investigation in largely unexplored areas. This is especially the case as this work has come before any significant body of expertise has evolved in the debugging of concurrent programs, let alone symbolic programs or problem-solving systems. This section discusses some of the lessons that we learned during the implementation of Poligon and, more importantly, during implementation of applications in Poligon. There are no great pearls of wisdom, but we hope that we can convey which of the features proved useful and which did not.

5.1. Simulation

Our first major observation was that simulation is hard and very time consuming but it is still easier than using real machines. This is due to the flexibility afforded by a simulator,¹ which allows the user to modify the topology, size, and behavior of the machines on which programs are to be run, and also to the inadequacy of the programming environments on existing parallel machines. Having poor development environments is not at all surprising given the comparative youth of these machines, but it was entirely a sufficient reason for not using them in our experiments. The fact that the tools that have been developed for multiprocessors tend to be designed for the debugging of C and FORTRAN programs means that these tools are of little or no use to Lisp programmers.

Because it is much easier to observe the internal behavior of a system in a simulator than on a real machine, we believe that simulation is likely to be an important aspect of program-

¹The CARE simulator is particularly good in this respect.

ming concurrent systems in the future. A good example of this is what happens when the program dumps you in the debugger.

- On a real parallel machine this presents significant problems. For instance, there is no way of immediately stopping all the processors. Even if the processor that finds the error broadcasts a halt message to the rest of the machine, a considerable amount of extra processing might have happened before the machine comes to rest. This can only confuse things.
- The technology for running debuggers in multiple stack groups on uniprocessors is well developed. This is not necessarily the case with parallel machines. It is much harder to get reasonable behavior out of inspector-like tools that must give a representation of data at random points in the machine. To chase data structures, the inspector will have to make references to remote processors, which could be a problem since it requires a suitable protocol for remote data structure manipulation, even on a distributed memory machine.
- Monitoring message traffic or memory operations on a multiprocessor, although not technologically hard, is hard in practice. This is because it often requires special hardware modification and also because the results delivered from this monitoring is at the wrong level of abstraction for anyone other than the implementors of memory systems or of communications networks. On a uniprocessor running a simulator, monitoring at the appropriate level of abstraction is simple.
- Finally, it should be noted that a crucial aspect of a simulator is that you can modify the simulator itself. Redesigning and then building hardware is a time consuming process. If one can modify a simulator in order to give more debugging information then this, itself, justifies the use of simulation.

5.2. Low-Cost Emulation

An important aspect of Poligon is that it has an emulation mode, Oligon. In this emulation mode, the accuracy and instrumentation of the CARE simulator are given up in favor of an emulation that gives a reasonable facsimile of Poligon's semantics when it is running under CARE, and it does so without a great deal of the cost.

Oligon runs entirely within one stack group. A considerable amount of effort in CARE's simulation is spent in switching stack groups. Oligon does not have to do this because of the way it implements its futures. Oligon's futures look just like Poligon's futures to a Poligon program. Indeed, the user does not even have to recompile a program in order to switch between Oligon and Poligon modes. Internally, however, an Oligon future encapsulates a message that will deliver the value of the future when its method is invoked evaluated.

When an Oligon future is created, it is recorded in a queue of unsatisfied futures. When the user defutures a future by executing a strict operator, the message that will evaluate the future is sent and the future is side-effected with the value of that evaluation.

The serial mode has a simple scheduler that, when it has nothing better to do, executes the messages associated with futures. Thus, all the messages associated with each future are evaluated at some time, which in turn guarantees that the all slot updates and node creation operations will, in fact, happen. This is necessary because futures are used to implement the equivalent Poligon behavior for all interprocess messages in Oligon. The rate at which

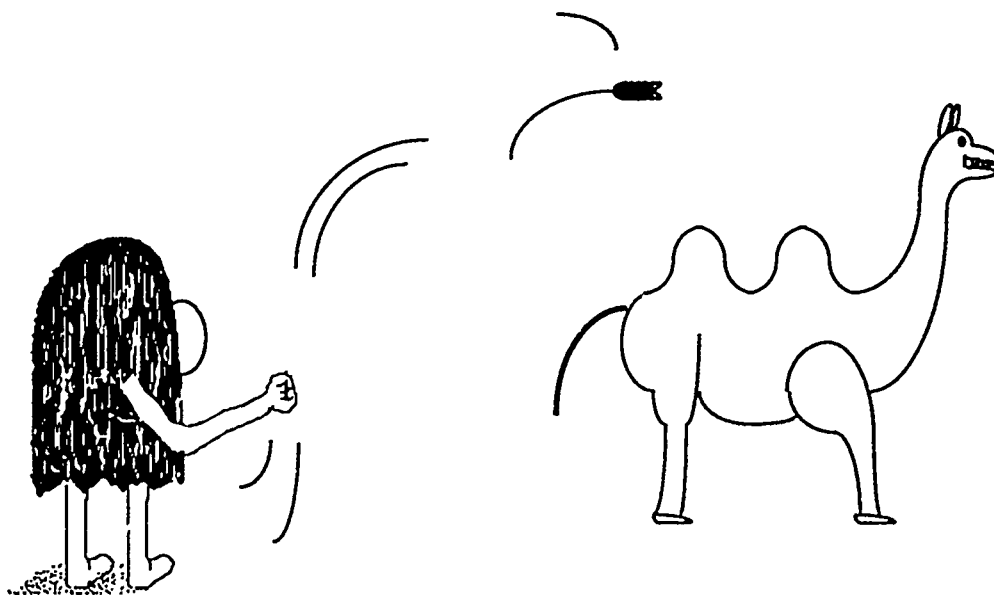
futures are forced by the scheduler is under the user's control, so that the user, to some extent, can emulate different levels of system load.

When a rule attempts to fire, instead of a new process being spun off for the context, a record that points to the context to be invoked is kept in the scheduler queue with all of the arguments that it would have been sent had it been operating in parallel. The scheduler loops around, removing events from this queue. The queue is implemented as a doubly linked list. This allows the user to tell the scheduler to operate in a number of different ways, selecting events to execute in a LIFO, FIFO, or random manner. The use of these different scheduling modes again allows the user to emulate Polygon running in its parallel mode under differing load conditions. The variations in the order in which the rules are in fact executed is sufficiently stressful to the application that once bugs have been eliminated in this emulated mode by means of different scheduler settings, the application will likely run in the Polygon mode without major incident. The sorts of events that are collected in this scheduler queue include the If parts, Action parts and Otherwise parts of rules.

5.3. Trace and Breakpoints

On trapping a lion in a desert [Petard 38]: The Wiener Tauberian method. We procure a tame lion, L_0 of class $L(-\infty, \infty)$, whose Fourier transform nowhere vanishes, and release it in the desert. L_0 then converges to our cage. By Wiener's General Tauberian Theorem, [Weiner 33a] any other lion, L (say), will then converge to the same cage. Alternatively, we can approximate arbitrarily closely to L by translating L_0 about the desert [Weiner 33b]

In the absence of any formal model for the debugging of concurrent blackboard systems, we found it necessary to include debug prints in our code. Although we have not gone much farther than this, we took the step forward that serial systems have made (and perhaps taken this to its logical conclusion) by developing facilities for tracing and breakpoints. It should be noted, however, that although these facilities are optimized for Polygon and the blackboard model, none of them in any way directly address the debugging problems of concurrent systems per se.



Breakpoint.

The native Lisp machines on which Poligon runs provide a number of trace and breakpoint facilities. These are not adequate for our purposes, not because they do not work (they do) but rather because their behavior is at the wrong level of abstraction. The Poligon compiler transforms the user's program into so many different functions and methods that putting trace or breakpoints on these is unlikely to be simple or worthwhile for the user. What the Poligon programmer would prefer is debugging facilities that are closely coupled both to the programming model of the system, and to the common mechanisms by which users introduce errors into their code.

A major problem with tracing activities in a real-time system is that any debugging code is likely to affect the behavior of the program itself. Indeed, it was our experience using MXA that leaving debugging code in is often simpler than trying to debug the real-time behavior of a program once the debugging code has been taken out. To this effect, Poligon tries hard to make its debugging facilities noninvasive. Whenever a trace or breakpoint is entered, the simulated real-time clock is stopped – another benefit of simulation – and started again on exit. Although the cost of executing the code that handles trace and breakpoints is not trivial, the perturbation caused by this code is far smaller than what would have been experienced if it were not possible to stop the clock during the actual processing of the trace. Formatting trace output is so expensive compared to the short evaluations in Poligon, which are typically less than a millisecond, that one cannot afford to count the cost of output of simulations.

Our attempt to provide more focused debugging was a four-pronged attack, first on the knowledge base and then on the blackboard, on general Poligon system activities, and finally on monitoring the program's parallel execution. It should be noted that at any point where a trace point can be applied in Poligon, a breakpoint can generally also be set where this is meaningful.

5.3.1. Debugging Rules

Poligon's rules are split up into a number of different components: the When, If, Select, Action, Otherwise, and Timeout parts. The evaluation of the rules takes place in the context of the set of definitions that have been evaluated up to the relevant point in the execution of the rules. Rules are grouped together in knowledge sources. Even though these are compiled out, in the sense that knowledge sources have no significance in the semantics of the program as it operates, it is still likely that the user will want to view all the rules in a knowledge source together. With a view to these issues we implemented a number of debugging features that are mentioned below and shown in Figure 5-1.

- Any trace or breakpoint operation that can be applied to a rule can also be applied to a knowledge source. This has the effect of applying that operation to all the rules in that knowledge source.
- All of the critical points in a rule are traceable. Thus, trace points can be set on the When, If, Select, Action, Otherwise, and Timeout parts of rules.
- Traces can be set so that the currently evaluated values of definitions are printed out at any point in a rule. This allows the user to monitor the behavior of a rule in terms of the definitions and when they are evaluated.
- Rule failure can be traced. It is a common feature of blackboard systems, and rule-based systems in general, that the user often does not know *why* a given rule fails to fire. The converse is often not the case because it is usually possible to set a breakpoint in a rule for when it does fire and one can then find out why it fired.

Because the user often does not know why a rule failed to fire, we implemented a facility in Polygon allowing the user to set traces on rules that are activated when a particular condition fails to pass. To do this the Polygon compiler takes advantage of the fact that the conditions of rules are usually the conjunction of a number of clauses. The compiler separates out these clauses, and at run time they are executed in the appropriate sequence, checking the trace settings as appropriate. If one of the clauses fails, it can execute the required trace. It is thus possible for the user to set a trace that says *Stop if this rule fails to fire because a clause fails after clause four in the If part.*

Knowledge Source :	Spot Threats
Rule :	Report Threatening Emitters
Knowledge Source :	Process Activities
Set flags for all rules in Process Redirected Observations	
Trace When part:.....	On Off
Trace When part failure on clause: NIL	
Trace If part:.....	On Off
Trace If part failure on clause:....	NIL
Trace Select part:.....	On Off
Trace Then part:.....	On Off
Trace Else part:.....	On Off
Trace Timeout part:.....	On Off
Break When part:.....	On Off
Break When part failure on clause: NIL	
Break If part:.....	On Off
Break If part failure on clause:....	NIL
Break Select part:.....	On Off
Break Then part:.....	On Off
Break Else part:.....	On Off
Break Timeout part:.....	On Off
Print Definitions:.....	Never When If Then Else Timeout
Abort [ABORT]	Do it [END]

Fig. 5-1. A menu showing the trace and break options available for rules and knowledge sources. In this example a knowledge source has been selected; any trace or breakpoints selected will apply to all rules in that knowledge source.

As is generally the case, these sort of trace features take a certain amount of computation to perform. This is incompatible with the goal of a high-performance system, so these traces are compiled out at high Optimize speed settings, a sacrifice of debugging ease for speed.

5.3.2. Debugging Using Nodes

A number of tracing features have been included within nodes. These in many ways mirror the behavior mentioned earlier for rules and knowledge sources. They are mentioned below and shown in Figures 5-2 and 5-3.

- It is possible to set traces on the reading, writing, or causing of an event on a slot.
- Just as it is possible to set traces for all rules in a knowledge source, it is possible to set a trace that will apply to all instances of a class, or simply to one particular node.

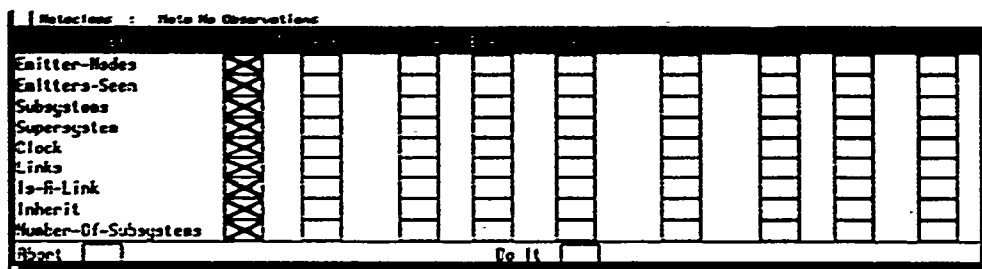


Fig. 5-2. Trace and break options available for operations on Poligon nodes. In this case the class Emitter has been selected. A similar menu allows all instances of a class to have these options set. Through this menu the user can set trace and breakpoints on system-defined slots, such as Number-Of-Subsystems, and on user-defined slots, such as Emitters-Seen.

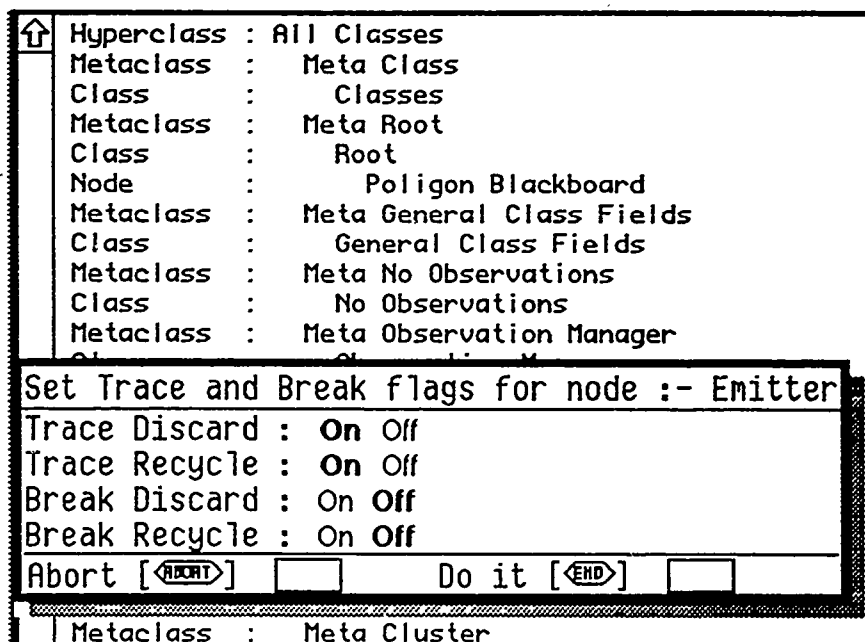


Fig. 5-3. A menu showing that it is possible to trace or break on the act of discarding or recycling a rule.

5.3.3. Tracing System Activities

Technical progress has merely provided us with more efficient means of going backwards.

— Aldous Huxley.

In addition to the trace and breakpoint features just mentioned, a number of trace features allow the user to monitor system functions. It is often the case that the user wants the system to progress to a certain point and then stop. This can be done because Poligon allows breakpoints to be set on the signal records that are read in, on the ticking of the clock, and on the creation of nodes. These are shown in Figure 5-4.

Please set these system parameters and user variables.		
<input checked="" type="checkbox"/> Trace Messages:.....	Verbose	Yes Yes-No Statistics No
<input type="checkbox"/> Trace Clock Ticks:.....	Yes	No
<input type="checkbox"/> Trace Signal Records:.....	Yes	No
<input type="checkbox"/> Break on Clock Ticks:.....	Yes	No
<input type="checkbox"/> Break on Signal Records:	Yes	No
<input type="checkbox"/> Trace Rules:.....	Verbose	Brief
<input type="checkbox"/> Trace Node Creation:.....	Yes	No
<input type="checkbox"/> Trace Message Punting:.....	Yes	No
<input type="checkbox"/> Break on Message Punting:.....	Yes	No

Fig. 5-4. A menu showing that it is possible to trace or break on the act of discarding or recycling a rule.

5.3.4. Monitoring the Parallel Execution of a Poligon Program

A number of additional trace features allow the user to optimize and debug Poligon programs:

- To allow the user to spot and rectify undesirable context punting, traces can be set.
- To allow the user to detect excessively slow pieces of code, the user's code is timed in the Oligon mode and a trace message can be emitted for any code fragment that requires more than a certain time to execute.
- To allow the user to detect slow code in the full, parallel case, Poligon allows the user to trace messages, by recording the messages and the arguments. It times the execution of the messages and allows the user to record only those taking a significant amount of time.
- To allow the user to get detailed information about the behavior of a program, Poligon is interfaced to the native machine's metering package. This allows the detailed metering of user code. Because the metering package can record only a short period of computation, the metering interface allows the user to specify a time to wait before metering commences. This allows the program to progress until it is actually running, as opposed to executing initialization code.

5.4. Perspectives

Finally, we would like to make an observation from our work on Poligon that has general applicability. A Poligon application is instantiated in a large number of data structures that owe their implementation primarily to the search for efficiency, not to intelligibility. It is often the case, therefore, that the programmer's cognitive model of the system may well be entirely different from the implementation model. Consequently, it is crucial to have tools that allow the user to view data structures in a manner that is consistent with the programming model, rather than the implementation model, if rapid debugging is to be possible. There are two simple examples of this in Poligon:

- Contexts have a lot of structure that is used by the system to implement their behavior. For implementation reasons, however, it was difficult to see the values of the definitions that are encapsulated within a context. Fortunately, we had developed an inspector tool that allowed data structures to be viewed simply from different viewpoints. It was therefore simple to define the default behavior for inspecting a context to display it as a mapping from the names of definitions to their values (see Figures 5-5 and 5-6). Another perspective allows the user to view contexts in

terms of their implementation rather than their purpose. Similarly, Polygon nodes are viewed by default in a manner that hides all system details, making it easier for the user to see what the program is really doing (see Figures 5-7 and 5-8). Being able to switch between multiple representations of the same data structure — and, of course, being able to implement these views easily — has proved to be of considerable utility.

#<Context 4/1 Assign Or Create Emitter>	
Definitions	
CREATED:	T
IS-IN-CACHE:	NIL
OBSERVATIONS-IN-TIMESLICE:	(0 3 8)
THE-EMITTER:	#<Future #<Future #<Remote Emitter 1 id=3>>>
THE-EMITTER-CACHE:	NIL
THE-EMITTER-ID:	3
THE-OBSERVATION-LOB:	91
THE-OBSERVATION-NODE:	NIL
THE-OBSERVATION-SITE:	(:BIG.EAR (9 67))
THE-OBSERVATION-TIME:	0
THE-OBSERVATION-TYPE:	:AI-B
Multiple Definitions	
PROCESS-OBSERVATIONS+ASSIGN-OR-CREATE-EMITTER+DEFINITIONS+THE-EMITTER+CREATED:	<i>unknown</i>
PROCESS-OBSERVATIONS+ASSIGN-OR-CREATE-EMITTER+DEFINITIONS+THE-OBSERVATION-TYPE+THE-OBSER	

Fig. 5-5. The default perspective for viewing contexts treats them as a means of mapping names into values.

#<Context 4/1 Assign Or Create Emitter>	
An object of flavor P::CONTEXT. Function is #<EQ-HASH-TABLE (Funcallable) 50232376>	
NUMBER:	4
P::TIMES-USED:	1
P::NUMBER-OF-TIMES-SCHEDULED:	0
P::OWNING-SITE:	NIL
P::LAST-VALUES:	NIL
P::AGENT-STACK-GROUP:	:NO-STACK-GROUP
CARE-USER::LOCALE:	NIL
CARE-USER::SELF=:	#<Remote Context 4/1 Assign Or Create Emitter>
CARE-USER::PENDING=:	unbound
CARE-USER::PENDING-TASK:	NIL
P::DEFINITIONS:	(<:PROCESS-OBSERVATIONS+ASSIGN-OR-CREATE-EMITTER+DEFINITIONS+THE-OI
P::RULE:	#<Assign Or Create Emitter>
P::NODE:	#<Remote Observation 4>
P::SLOT:	PU::REDIRECTED-FLAG
P::VALUE:	<Nil>
P::TRIGGERING-NODE:	#<Remote Observation 4>
P::EXPECTATION-ARGS:	NIL
P::THEN-PART:	(PU::PROCESS-OBSERVATIONS+ASSIGN-OR-CREATE-EMITTER+SELECT+CASE 0)
P::CASE-PART:	2
P::OTHERWISE-PART:	NIL
P::TIMEOUT-PART:	NIL
P::INDIRECT-TO:	NIL
P::TAG:	:NEWNODE-CREATED
P::CHECKED-TAG:	2
P::ALL-ACTION-PARTS:	2
P::RULE-SHOULD-BE-ACTIVATED:	T

Fig. 5-6. An alternate perspective for viewing contexts allows them to be seen in terms of their implementation.

- In a naïve environment Polygon's implementation makes navigating over the network of Polygon objects very difficult. This is because Polygon's nodes are viewed as remote addresses. These remote addresses point to streams, which in turn point to processes. Somewhere in the context of these processes is some pointer to the actual object that is primarily associated with the process. A large number of mouse clicks in an inspector would therefore be required to get from the remote address of a node to the node that it really points to. Again, fortunately, we had developed tools that allowed us to decouple the printed representation of our data structures from their mouse-sensitive values. Thus, a remote address to a node might be printed as #<Remote Aircraft-42>. The name Aircraft-42, however,

would be mouse-sensitive, and when clicked on, would deliver the node we were interested in. The machine is left to do all the hard work of figuring out what the user wanted to see, a huge saving of effort.

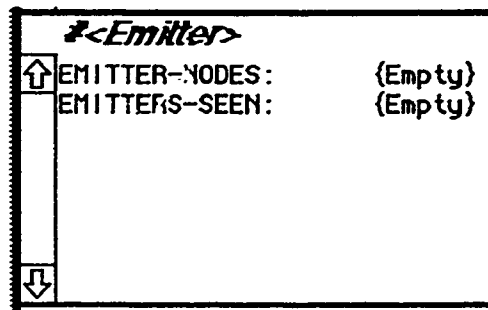


Fig. 5-7. The default perspective for inspecting Polygon nodes causes only user slots to be visible.

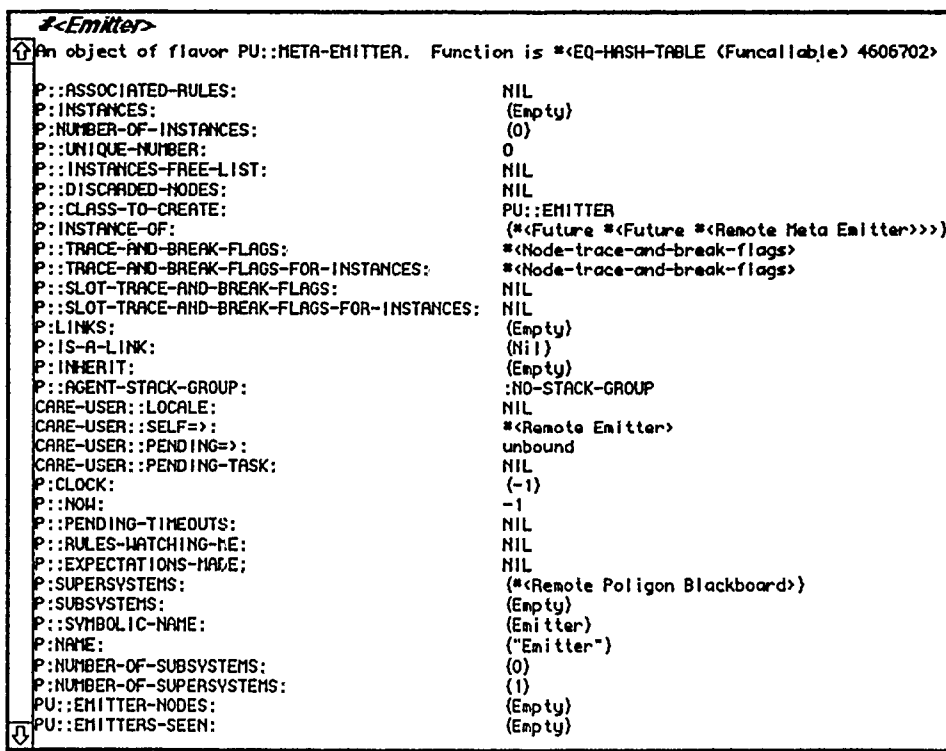


Fig. 5-8. An alternate perspective for viewing Polygon nodes allows the programmer to see the entire system-defined structure of nodes.

5.5. Compiler Optimization

A significant factor in our ability to develop and debug Polygon programs seems to have been the controlled introduction of compiler optimization during debugging. The Polygon programmer has available a number of debugging aids that are progressively switched off as the user asks the compiler for higher levels of optimization. This seems to have been a good decision. The sorts of transformations that are applied to programs, even in conventional, serial systems, can be somewhat counterintuitive and confusing in the process of debugging a program. Clearly, these optimizations should not affect the semantics of a correct program, so they are only of significance in the presence of program bugs. A number of the optimizations that the Polygon system uses have been discussed above in the re-

lated sections. Our intention here is to reiterate our belief in the importance of this design strategy.

6. Conclusions

There is altogether no lack in Genesis of retribution for failure to obey the Lord. It would not seem, however, that the examples made had much effect. We are thus driven to the conclusion that the direct incentive is more effective than the disincentive, the carrot more useful than the stick. A possible explanation of this fact might be based on the theory that the wrong donkey is beaten every time.

— C. Northcote Parkinson, *Incentives and Penalties*.

In this paper we have attempted to detail the design and implementation of Poligon, a concurrent problem-solving system modeled closely on the blackboard metaphor.

A number of papers concerning Poligon have focused on its architecture, motivations for its design, its performance, and experiments performed on Poligon applications. None of these publications have indicated how we implemented it or the obstacles encountered and the mistakes we made along the way. This paper described the implementation in sufficient detail that the reader should be able, given enough effort, to implement a system with similar, or better behavior and performance.

We concentrated on Poligon's design as an example of an attempt to develop a high performance, concurrent problem-solving system. We have delineated a set of issues that implementors must address in order to achieve good performance in a concurrent blackboard system. Many of these observations are applicable to other architectures and to serial systems as well. The important aspects are node creation, knowledge search, conflict resolution, knowledge invocation, context evaluation, slot reads, slot updates, event posting, and the efficient handling of stack groups and processes. Each of these aspects of a system's performance were discussed with particular reference to the Poligon model.

A number of features in Poligon proved to be inadequate, difficult, or didn't work at all. Among these were run-time property inheritance, node deletion and reuse, message passing in a rule-based system, the efficient use of pipelines, and load balancing. We also found ourselves unable to shield the user from the differing costs of communicating with local versus remote memory.

We have found that the blackboard model, an appealing cognitive model for concurrent problem solving, does not necessarily work as well in practice as intuition might lead one to expect. As a consequence, although we hoped to deliver many orders of magnitude of speedup due to parallelism, we have only been able to show about one order of magnitude and there are indications that this might scale to about two orders of magnitude. When comparing our application against the same application written in AGE, however, we observe that the application's simulated performance in Poligon was about fifteen thousand times faster. At least by comparison, therefore, we can assert that we have, indeed, built a high-performance concurrent blackboard tool.

In countries with an aristocratic tradition (like Britain) the highest status is associated with official position, birth, education, athletic prowess and gallantry in battle. In countries without any such tradition (like U.S.A.) the highest status is associated with the biggest capital and income. Very seldom do we meet a millionaire with a V.C., and Sir Thomas More's achievement, in being both knighted and canonized, is likely to remain an unbeaten record.

— C. Northcote Parkinson, *Incentives and Penalties*.

7. Bibliography

- [Aiello 86] Aiello, Nelleke. *User-Directed Control of Parallelism: The Cage System*. Technical Report KSL-86-31, Knowledge Systems Laboratory, Computer Science Department, Stanford University, April 1986.
- [Aiello 88] Aiello, Nelleke. *Cage: The Performance of a Concurrent Blackboard Environment*. Technical Report KSL-88-80, Heuristic Programming Project, Computer Science Department, Stanford University, 1988.
- [Bandini 89] Bandini, Jean-Christophe. *Poligon Applications*. Technical Report KSL-89-43, Heuristic Programming Project, Computer Science Department, Stanford University, 1989.
- [Brown 86] Brown, Harold, Eric Schoen, and Bruce A. Delagi. *An Experiment in Knowledge-Base Signal Understanding Using Parallel Architectures*. Technical Report STAN-CS-86-1136, Heuristic Programming Project, Computer Science Department, Stanford University, 1986.
- [Byrd 88] Byrd, Gregory T. and Bruce A. Delagi. *A Performance Comparison of Shared Variables vs. Message Passing*. Technical Report KSL-88-10, Heuristic Programming Project, Computer Science Department, Stanford University, 1988. Also in Proceedings of Third International Conference on Supercomputing, Vol. 1, pages 1-7, Boston, MA, March 1988 International Supercomputing Institute.
- [Delagi 86] Delagi, Bruce A., Nakul P. Saraiya and Gregory T. Byrd. *LAMINA: CARE Applications Interface*. Technical Report KSL-86-76, Knowledge Systems Laboratory, Computer Science Department, Stanford University, 1986.
- [Delagi 88a] Delagi, Bruce A., Nakul P. Saraiya, Gregory T. Byrd, and Sayuri Nishimura. *CARE User's Manual*. Technical Report KSL-88-53, Heuristic Programming Project, Computer Science Department, Stanford University, 1988.
- [Delagi 88b] Bruce A. Delagi and Nakul P. Saraiya. *ELINT in LAMINA: Application of a Concurrent Object Language*. Technical Report KSL-88-33, Heuristic Programming Project, Computer Science Department, Stanford University, 1988. Also in SIGPLAN Notices, February 1989.

- [Erman 80] Erman, Lee D., Frederick Hayes-Roth, Victor R. Lesser, D. Raj Reddy. *The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty*. ACM Computing Survey. 12: 213-253. June 1980.
- [Engelmore 88] Engelmore, Robert and Tony Morgan, eds. *Blackboard Systems*. Addison-Wesley Publishing Company Inc., Menlo Park, CA 1988.
- [Forgy 76] Forgy, C. and J. McDermott. *The OPS Reference Manual*. Carnegie-Mellon University. 1976.
- [Gupta 86] Gupta Anoop. *Parallelism in Production Systems*. Technical Report, Computer Science Department, Carnegie-Mellon University, March, 1986. Ph. D. dissertation.
- [Halstead 84] Halstead, R. H. Jr. *Implementation of Multilisp: Lisp on a Multiprocessor*. Proceedings of the ACM Symposium on Lisp and Functional Programming. 9-17, August 1984.
- [Hayes-Roth 85] Hayes-Roth, B. *Blackboard Architecture for Control*. Journal of Artificial Intelligence. 26: 251-321. 1985.
- [Hillis 85] Hillis, W. D. *The Connection Machine*. MIT Press. Cambridge, MA. 1985.
- [Nii 79] Nii, H. Penny, and Nelleke Aiello. *AGE: A Knowledge-based Program for Building Knowledge-based Programs*. Proceedings of the 6th International Joint Conference on Artificial Intelligence: 645-655, 1979.
- [Nii 82] Nii, H. P., E. A. Feigenbaum, J. J. Anton, and A. J. Rockmore. *Signal-to-Symbol Transformation: HASPISIAP Case Study*. Technical Report HPP-82-6, Heuristic Programming Project, Computer Science Department, Stanford University, 1982. Also in AI Magazine. 3:2, 23-35, 1982.
- [Nii 86] Nii, H. Penny. *Blackboard Systems*. Technical Report KSL-86-18, Knowledge Systems Laboratory, Computer Science Department, Stanford University, April 1986. Also in AI Magazine, 7:2 and vol. 7:3, 1986.
- [Nii 88a] Nii, H. Penny, Nelleke Aiello and James Rice. *Frameworks for Concurrent Problem Solving: A Report on Cage and Polygon*. Technical Report KSL-88-02, Knowledge Systems Laboratory, Computer Science Department, Stanford University, April 1988. Also in [Engelmore 88].
- [Nii 88b] H. Penny Nii, Nelleke Aiello, James Rice. *Experiments on Cage and Polygon: Measuring the performance of Parallel Blackboard Systems*. Technical Report KSL-88-66, Heuristic Programming Project, Computer Science Department, Stanford University, 1988. Also in *Distributed Artificial Intelligence II*. L. Gasser and M. N. Huhns (eds). Morgan Kaufmann, San Mateo, CA 1989.

- [Osgood 28] W. F. Osgood. *Lehrbuch der Funktionentheorie*, vol 1 (1928) p 178.
- [Petard 38] H. Petard, "A Contribution to the Mathematical Theory of Big Game Hunting," *American Mathematical Monthly* 45:446, 1938.
- [Rice 84] Rice, James. *The MXA user's and writer's companion*. Systems Programming Ltd., The Charter Abingdon, Oxon, U.K. 1984.
- [Rice 86] Rice, James. *The Poligon User's Manual*. Technical Report KSL-86-10, Heuristic Programming Project, Computer Science Department, Stanford University, 1986.
- [Rice 88a] Rice, James. *Problems with Problem-Solving in Parallel: The Poligon System*. Technical Report KSL-88-04, Heuristic Programming Project, Computer Science Department, Stanford University, 1988. Also in Proceedings of Third International Conference on Supercomputing, pages 25-34, Boston, MA, March 1988 International Supercomputing Institute, and *Artificial Intelligence, Simulation and Modelling*, Lawrence Widman (ed), John Wiley Publishing Company, New York 1989.
- [Rice 88b] Rice, James. *The Elint Application on Poligon: The Architecture and Performance of a Concurrent Blackboard System*. Technical Report KSL-88-69, Heuristic Programming Project, Computer Science Department, Stanford University, 1988. Also in Proceedings of International Joint Conference on Artificial Intelligence, Vol. 1, 212-217, 1989.
- [Rice 88c] Rice, James. *The Advanced Architectures Project*. Technical Report KSL-88-71, Heuristic Programming Project, Computer Science Department, Stanford University, 1988.
- [Saraiya 89] Saraiya, Nakul P. *Design and Performance Evaluation of a Parallel Report Integration System*. Technical Report KSL-89-16, Heuristic Programming Project, Computer Science Department, Stanford University, April 1989.
- [Schoen 86] Schoen, Eric. *The CAOS System*. Technical Report KSL-86-22, Heuristic Programming Project, Computer Science Department, Stanford University, 1986. Also in Proceedings of DARPA Expert Systems Workshop, 160-170, April 1986.
- [Weiner 33a] Weiner, Norbert. *The Fourier Integral and Certain of Its Applications* (1933) pp 73-74.
- [Weiner 33b] Weiner, Norbert. *The Fourier Integral and Certain of Its Applications* (1933) p 89.

My idea of an agreeable person is a person who agrees with me.

— Benjamin Disraeli

AIDE A Distributed Environment for Design and Simulation

**by
Nakul P. Saraiya**

**KNOWLEDGE SYSTEMS LABORATORY
Department of Computer Science
Stanford University
Stanford, California 94305**

This Working Paper is intended for use only within the KSL, and has not been approved for general distribution. It contains information that may be incomplete, preliminary, or not reviewed according to standard procedures. External distribution can only be arranged by contacting the author.

AIDE
A Distributed Environment
for
Design and Simulation

Nakul P. Saraiya

MSAI Practicum Report

June, 1986

Support for this work was provided by the following . DARPA/RADC, under contract F30602-85-C-0012, NASA, under contract number NCC 2-220, Boeing Computer Services, under contract number W-266875.

Abstract

AIDE is an environment that provides facilities for the hierarchical specification and simulation of systems. In addition, a user of AIDE can distribute a simulation over a network of computers. Achievable concurrency in a distributed simulation depends on the functional characteristics of the system being simulated and on the ability of the simulator to exploit its knowledge of these. AIDE can use the information contained in the structural and behavioral specification of a system to increase concurrency and decrease synchronization costs during distributed simulation. Performance analyses of the AIDE distributed simulation algorithm for a simulated multiprocessor architecture indicate that (1) the disparity between communication costs and event-processing costs and (2) load imbalance can significantly limit speedup when concurrency is available. The distributed aspects of the AIDE environment are implemented through an extension of the underlying object-oriented programming system (FLAVORS) to multiple machines.

1. Introduction

A design system is expected to provide a *framework* for a designer to adequately implement representations of certain interesting physical or abstract entities that perform some function. In doing so, it must provide a suitably precise formalism and an integrated set of tools, thus allowing the designer to conveniently specify, modify and evaluate such representations [2, 12]. AIDE¹ is an attempt to provide such a framework.

AIDE evolved in the context of the Advanced Architectures for Expert Systems project of the Heuristic Programming Project. One of the goals of this project is to specify a concurrent, distributed-memory computer architecture that will speed up AI programs through parallel processing. AIDE grew from the need to implement and evaluate this architecture, called CARE [4], flexibly and efficiently. As the evaluation was to be done by simulating the execution of large application programs (for example, CAOS, POLIGON, and ELINT), this led to investigating the utility of distributed simulation both as a means of reducing simulation turnaround time and in ensuring that the simulated machine was being programmed fairly, that is, without making use of the *real* shared memory available on the host machine.

This document describes the essential aspects of AIDE. The first part of the document concerns design representation and capture. We briefly describe the facilities that AIDE provides for the hierarchical specification of systems. The second part of the report deals with the functional verification and performance evaluation of system specifications through simulation. We concentrate on describing what has been the major portion of this work, namely, the investigation of distributed simulation. The appendix discusses the extensions to the underlying programming system that form the basis for the distributed system features of AIDE. More detailed documentation for the system is contained in the user's manual [10].

2. Design Capture

Design capture denotes the process by which a designer specifies a representation of an abstract entity (that may be physically realizable) to a design system. The design system must provide a representation formalism that is sufficiently general and expressive to describe a wide variety of such entities. It must also allow the designer to conveniently use this formalism. In this section, we discuss the representation formalism used in AIDE, and describe the facilities it provides for design capture.

¹AIDE is a Distributed Environment.

2.1. Design Representation

Every real-world or abstract entity may be thought of in terms of its structure and its behavior. A structural view of an entity is any organizational view of the entity that decomposes it into (functionally or otherwise) semi-independent components. A behavioral view of an entity is a conceptual formalization of the way certain interesting properties of the entity change over time. Of course, structural and behavioral formalizations of an entity complement each other in attempting to define the modelled entity. Different formulations of structure and behavior emphasize different qualities of the entity. The representation of an abstract entity in terms of particular aspects of its structure and behavior constitutes a design or model in AIDE; the system provides a formalism that allows the user to describe such a design (entity) by describing its structure and its behavior.

Design is a creative and incremental activity; designs evolve over time in the designer's mind. For example, it is well-known that the process of design is "partially-structured" [2] in that designers often work both top-down and bottom-up. A design system must provide a representational vocabulary that recognizes this and a capture mechanism that makes such an approach to design convenient.

AIDE follows PALLADIO [2] and HELIOS [5] in using an encapsulation of structure and behavior called a component as the fundamental unit of design. It allows the user to combine and refine components in well-defined ways during design capture. A design (description) at any instant in time is exactly such a collection of related components.

2.1.1. Hierarchical Partitioning

Hierarchical decomposition is one of the ways in which a designer works top-down to make the process of designing a complex system more tractable. PALLADIO viewed the process of circuit design as the incremental refinement of a functional description of the circuit into its physical realization. Here the basic design refinement step was partitioning the circuit at some abstract structural level into constituent components specified at either the same level or a less abstract level.

AIDE supports hierarchical partitioning directly and simply by allowing the user to define a component to be the composition of arbitrary (perhaps incompletely specified) subcomponents.

2.1.2. Design Libraries

Complementing hierarchical partitioning is the use of *prototypes* to build on previous work [5, 2]. This allows the designer to rapidly create new designs by modifying existing designs or by applying new composition rules to them. AIDE supports this idea through the use of *libraries*, which are collections of prototypical components that may be stored between sessions and copied or re-used in the creation of new components.

2.1.3. Behavior

Component behavior specifications must be efficient both in expression and during simulation. AIDE uses the ZETALISP [13] language and programming environment directly in addressing both these concerns, paying the penalty of expecting the user to be a reasonably competent LISP programmer. AIDE defines the behavior of a component that is being modelled as the composition of other components to be the composition of the behaviors of the individual subcomponents.

2.1.4. Implementation

It is natural to use the object-oriented programming paradigm to implement the components that represent a design, directly mapping from entities in some "real" world (of the designer's choosing) to the data objects manipulated by the design system. AIDE uses the object-oriented programming facilities provided by the FLAVOR system [13]. Every component is implemented as an instance of some

component class, where the class defines a component type and is internally implemented as a *flavor*.²

From this point on, we intentionally blur the distinction between the representation of a design and the system's implementation of that representation. We hope that this approach will assist in the description of the design system by providing implicit operational definitions of the various abstract terms (like "structure" and "component") used by the representational formalism.

2.2. Structure

As was stated above, a design in AIDE consists of a specification of its intended structure and behavior. In this section, we discuss the structural aspects of such a *design description* in more detail. This discussion is in terms of the unit of a design description, namely, the component.

To the design system, a component's structure consists of two parts :

- the component's own properties, and,
- the component's relationships with other components.

2.2.1. Component Properties

The designer sees a component as a "black box" of a particular type that has a collection of local named attributes with associated values. The allowable attributes of a component are defined by its type, while the *values* on these attributes may (and usually do) differ for each component instance. A subset of these properties³, the *state* properties, are used to generate the behavior of the component. Special state properties known as *ports* (input and output) constitute a component's interface to its environment. Other properties are used by AIDE to maintain and display components.⁴

AIDE provides the *defcomponent* form for a designer to define the structural properties of a new component type. It has a graphical editor to capture and alter the display properties held by components.⁵

Figure 2-1 is a simple example of the component class declaration for an abstracted D-type flip-flop. Each instance of *d-flip-flop* has three input ports (named *d*, *clock*, and *clear*), one output port (named *q*), and no internal state.

```
(defcomponent D-Flip-Flop
  (:input D Clock Clear)
  (:output Q)
  (:documentation "Class of positive-edge-triggered D-type
    flip-flop with direct clear. Uses 'high', 'low' and 'x'
    logic signals. Has unit delay between an input transition
    and stable output."))
```

Figure 2-1: Definition of the *d-flip-flop* Component Class

²Within the usual inheritance network.

³We use the term "properties" loosely to mean the collection of attributes and their values.

⁴These are automatically inherited by every component class.

⁵A large part of the graphical interface was modelled after that used by HELIOS and PALLADIO.

For a complete description of the `defcomponent` form see [10]; suffice it to say here that it translates into the appropriate FLAVORS declarations, resulting in the definition of a new component class being present in the environment. Thereafter, components of this class or type may be created by the usual instantiation mechanism.

2.2.2. Structural Relationships

The second part of the structural description of a component is a description of its relationships with other components. There are two structural relationships that may hold between components :

- **Composition.** Any component may be a subcomponent of exactly one component and every component may be composed of any number of subcomponents. When a component is *composite* (made up of subcomponents), it may share its ports, for behavioral purposes, with those of its subparts through the "connection" relation.
- **Connection.** This relation holds between individual ports of two components and is specified by lines which *connect* the relevant ports. Lines may connect an output port of some component to an input port of another component except when connecting ports between a composite component and one of its subcomponents, in which case the connected ports are of the same type (port sharing). Usually a line connects just two ports; **contacts** are special entities that provide fan-in and fan-out capabilities for lines.

These structural relationships between individual components are captured by AIDE through its graphical structure editor.

2.2.3. Prototypes

Traditionally, object (frame) systems have had difficulty in implementing a general mechanism for capturing complex relationships that must hold between sets of instances of various classes. The "connection" structural relation is one such relation. It is difficult to declare this information in the class definition of a composite component, although it is convenient to visualize the component as a group of connected subcomponents. For example, a processing *site* in the CARE architectural model consists of an evaluator, an operator, and a number of *fifo-buffers*, *net-inputs* and *net-outputs*. Though these are all connected in a well-defined manner, it is extremely tedious to specify this information in the `defcomponent` declaration for a *site*.

The solution we have adopted in AIDE is to store connectivity information about a composite component type as a "canonical" instance of the relevant component class. This canonical instance is called the **prototype** of its class and it is created using the design editor, it may then be stored in a design library. The structure of a component class is fully specified by the existence (in the environment) of both a `defcomponent` declaration and a prototype.

2.2.4. The Design Editor

A component in AIDE may be accessed through the graphics-based, menu-driven interface which provides operations for viewing and selecting components. Top-level components or *devices* are maintained in book-keeping entities or *worlds*, each of which may have several windows or *viewports* viewing the relevant device. The design editor uses the graphics-based interface in providing operations to create new devices and edit their structure. Menu-driven commands allow the user to create, alter and delete components, lines, ports and contacts (see Figure 2-2). There are also facilities to copy devices into permanent file storage, "prototize" devices for inclusion in libraries (see Section 2.2.3), and load devices and libraries from file. A complete description of the operations provided by the editor may be found in [10].

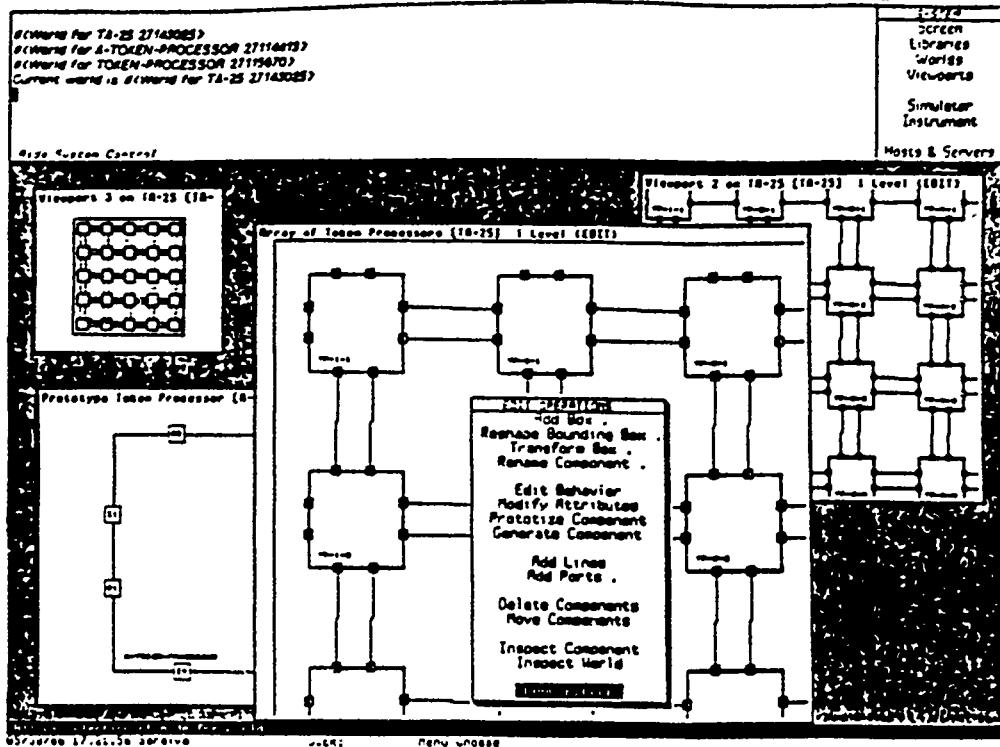


Figure 2-2: The AIDE Structural Editor

2.3. Behavior

The behavior of a component is the interaction of the component with its environment over (abstracted) time. A *behavioral specification* in AIDE applies to a class of component. It defines the interaction between a component of that class with a behavioral *simulator* to *generate* the requisite behavior over time. Since the simulator in AIDE is event-driven, this interaction takes the form of the consumption and production of *events*, which are encapsulations of the *state changes* in the simulated system. A behavioral specification for a component is therefore simply a specification that relates changes in the values of input ports with changes in the values of output ports over (simulated) time. Components whose output values depend on a history of input values make use of their internal state properties.

AIDE provides the *defbehavior* form to capture the behavioral specification of a component class. A behavioral specification is implemented by a *method* on the component class. All the state properties of a component are accessible to this behavioral method.

During the execution of a simulation, events pertaining to a component are *consumed* or processed when the simulator propagates the encapsulated state change and then invokes the relevant component's behavior method with the event as a parameter. Within a behavior method invocation, the simulator may be informed of new events through a call to the *assert* function to specify a change that will be true of some state of the component at some future simulated time. This results in new events being *produced* or generated.

2.3.1. An Example

Figure 2-3 is an abstract behavioral specification for the d-flip-flop component class. The signal on the d input is transferred to the q output when the clock input goes from low to high. If, however, the clear input goes low, then so does the q output. The q output is unaffected by the d input whenever the clock is stable. The clock period is two simulated time units, and input setup time is ignored.

```
(defbehavior D-Flip-Flop (component state signal now)
  ;; The above 4 parameters "destructure" the event.
  ;;
  ;; Clear Clock D | Q
  ;; -----
  ;; low  x    x  | low
  ;; high ↑    high | high
  ;; high ↑    low  | low
  ;; high low  x   | Q0
  ;;
  (selectq state ; The state attribute changed by this event.
    (Clock ; Clock just changed.
      (when (eq (state-value (port-signal Clear)) 'high) ; Clear is high.
        (when (eq signal 'high) ; Clock is rising.
          (when (< (- now (state-time (port-signal D))) ; D changed in
            2) ; this clock cycle.
            (Assert Q (state-value (port-signal D)) ; Transfer D to
              (1+ now)))))) ; Q after unit delay.
      (Clear ; Clear just changed...
        (when (eq signal 'low) ; ...& went low.
          (Assert Q 'low (1+ now)))))) ; Q goes low after unit delay.
```

Figure 2-3: Behavior Declaration for the d-flip-flop Class

There are some points worth noting in the example of Figure 2-3.

- The style illustrates one of the benefits of event-driven simulation : only the state *changes* are propagated as opposed to recomputing the state of the entire system at every step [1].
- The declaration has an explicit notion of the passage of time. Simulated time units have user-defined semantics. It is up to the user to ensure that the units are used consistently by different connected components.
- The state changes specified by the events for a given simulated time are *all* made before behavior methods are invoked on the events.⁶ Hence, there is no need to specify a clause to handle a change in d occurring at the same simulated time as a clock transition from low to high, where the clock event is "processed" earlier in real time than the d event.

2.3.2. Composite Behavior

The benefits to be gained by hierarchical simulation are well-known. Once the behavior of a multi-component system is verified, the designer may reduce simulation turnaround time by abstracting this behavior into a less detailed behavior (specification) that realizes the same function. Conversely, a designer may want to do detailed simulation of a component after doing an initial high-level simulation of its functionality in order to determine parameters and trade-offs at the next lower structural level.

AIDE directly supports hierarchical simulation by allowing a designer to specify whether a composite

⁶This excludes zero-delay events generated by the behavior methods. The special case of zero-delay events on the local state of a component is allowed by AIDE. A behavioral specification that uses this feature must handle such state changes actually occurring at the time the zero-delay event is generated.

component's behavior is its own defined behavior (*top-level*) or the compounded behaviors of its connected subcomponents (*internal*). For example, if we designed a shift-register from D-type flip-flops, we might initially verify the design using the internal behavior of the shift-register, that is, the composite behavior of its flip-flops. Later, when using a shift-register in the design of a control-unit, we might use a top-level characterization of its functionality for efficiency. Thereafter, when modelling the control-unit as realized in silicon, we may revert to the internal (flip-flop) behavioral perspective of the shift-register.

How does composite behavior work? During simulation, events on output ports are immediately transformed into events on the furthest participating connected input ports (if any).⁷ These are then forwarded by the terminal port to the simulator to be consumed by the terminal component at the specified simulated time.⁸ Hence, the effects of a local change propagate through the system along connection paths, achieving the required overall system behavior.

2.3.3. Behavior Requirements

To ensure that a composite system's behavior is predictable from that of its parts, a top-level behavioral specification is usually required to satisfy the following properties [3, 6] :

1. **Functional Output.** Events generated on output ports of a component depend only on events consumed on its input ports and internal states, and on its initial state. This implies that a component's top-level behavior should only depend on those aspects of its environment that are visible as changes on its input ports.
2. **Realizability.** An event generated for simulated time t cannot be affected by events consumed by the component for simulated times greater than t . This simply reflects the notion that no real system can predict the future.
3. **Positive Delay.** An event on an input port or internal state with simulated time t can only generate events on output ports with simulated time greater than t . This captures the idea that no real system can alter the past.

Note that this excludes zero-delay events between components. However, we allow zero-delay events on the internal states of components, as long as there is a finite delay before such events result in output events.

A quick inspection of Figure 2-3 should verify that the behavior specified for d-flip-flop satisfies these properties.

⁷ Consider the device just before the execution of a simulation. The structural hierarchy of components and subcomponents forms a tree rooted at the device. This tree can be flattened into a "participation digraph" as follows. The set of nodes of this digraph is generated by doing a breadth-first traversal of the tree during which any encountered component that is using top-level behavior is added to the node set and the subtree rooted at this component not traversed thereafter. The arcs of the digraph are then formed by following the outward connecting lines from all output ports of such participating components until an input port on some participating component is reached. Such input ports are the furthest participating connected input ports or *targets* of their respective participating output ports.

⁸ Event transformation is done cooperatively by the ports themselves through message-passing. Ports are implemented as flavor-instances. Output ports can cache their target input ports since structure is static for a simulation. The `assert` macro results in a message being sent to the output port to inform the simulator of the new event. The output port simply forwards this message to its cached target, which carries out the actual interaction with the simulator.

3. Design Validation

Once a design has been specified to a design system, the designer must be able to validate it by ensuring that it meets both its functional and performance goals. In the absence of formal verification methods, simulation is a common technique to establish the *functionality* of a design [11, 7]. Furthermore, since simulation, unlike emulation, automatically carries with it an explicit notion of time⁹ it can also be used to compare the *performance* of a design with other designs or real systems that realize the same function.¹⁰ The performance evaluation of a design is often as important to the designer as verifying its functionality [3, 7].

In this section we look at discrete event simulation in AIDE. After a brief discussion of sequential simulation, we define the distributed simulation problem and present the approach used in AIDE to solve this problem. Finally, we present some experimental results on the performance of the AIDE distributed simulation scheme for a simulated multiprocessor architecture based on CARE.

3.1. Discrete Event Simulation

While there are various types of simulation (see [8] for a good characterization of simulation methods), we are concerned here only with discrete-time, event-driven simulation. Our choice of event-driven simulation is based on the fact that most designs exhibit very few state changes at any given time; furthermore, such state changes are widely scattered in time, especially in "mixed-mode" simulations. Hence it is more efficient to keep track of the changes and their times rather than recompute all states at every instant in time. Before proceeding with our discussion, it is useful to consider some definitions.

3.1.1. Consistency and Acceptability

An **event** is an atomic state change in the simulated system during the execution of a simulation. It may be represented as a record consisting of (1) a component, (2) the port or internal state of the component that changes, (3) the value that it gets, and (4) the simulated time of this change. Two such events are equivalent if they represent the same state change to the simulated system, though for different executions of the simulation.

Simulated time is the designer's abstraction of real time. The state of the real system (device) at any real time corresponds to the state of the simulated system (device) at the corresponding simulated time [8]. Simulated time takes on non-negative, discrete, and, for convenience, integer values.

The **simulation** of a component (device) refers to the *execution* of a simulation of a component (device) under the control of some simulation algorithm which regulates the consumption (and hence, production) of events relevant to that component (device) over real time. For a given simulation there is an associated set of events. We say that two simulations are equivalent if they produce equivalent event sets (given that the device being simulated is deterministic). Two simulation algorithms are **consistent** if any two simulations under the control of each algorithm, respectively, are equivalent. The actions of a simulator to achieve consistency (using a simulation algorithm) are collectively called **synchronization**, hence the algorithm is often called a *synchronization algorithm*.

Lastly, we call a synchronization algorithm **acceptable** if it is consistent with itself and if it accurately reflects the behavioral specification of the simulated system. Intuitively, this means that a synchronization algorithm is acceptable if it always generates all and only those events induced by the initial state of the simulated system (including the initial events) and the behaviors of the components being simulated.

⁹As construed by the designer.

¹⁰Arvind et al, in [1], characterize emulation as an abstraction of simulation in which timing detail is compiled away.

3.1.2. Synchronisation

Acceptability is the goal of every synchronization algorithm. Since almost every implementation of a simulator (including that of ADE) depends on side-effects to changeable state¹¹, acceptability operationally means that the simulation algorithm must *control the consumption of events during execution so that behavior-generating code is invoked in the correct context*. (This is not necessarily the case; for example, a simulation system that uses a strict logic programming system to implement structural and behavioral specifications need not concern itself with this issue since all "state changes" will persist in such a system. Of course, the burden of storing and managing this information has now been thrust upon the logic programming system.) With this implementation model in mind, we provide below an informal relation on events that will be useful in analyzing the acceptability of synchronization algorithms.

An event e_i **preempts** another event e_j if either of the following is true :

1. e_i and e_j specify a change to the same state entity but the simulated time of e_i is greater than the simulated time of e_j ;
2. the state change specified by e_i overwrites information that is used by e_j and the behavior of the relevant component to generate an event.

Two events are **independent** if neither preempts the other.

We claim that an acceptable simulation algorithm is one that generates an event set such that for every e_i and e_j in the set, if e_i preempts e_j then e_i is consumed after e_j .

In theory a simulator has to run the entire simulation to determine the set of preemption relationships on its event set; in practice, however, it computes a set of *possible* event preemptions. The requirement is that this set be a superset of the set of actual event preemptions. The problem of synchronization (distributed or otherwise) is therefore essentially the problem of dynamically determining potential event preemptions and processing those events that cannot be preempted.

3.2. Sequential Simulation

We discuss briefly the mechanism by which sequential simulation works in ADE.

3.2.1. Synchronization Using Simulated Time

The standard sequential synchronization algorithm makes use of the simulated time of an event and the realizability requirement on component behavior (see Section 2.3.3) to achieve acceptability. Events with lower simulated times are always processed (consumed) before events with higher simulated times; therefore, at the time an event is processed, all the events that could possibly have preempted it have already been consumed.

The main advantage of this synchronization algorithm is its simplicity. It is easily implemented on a serial machine. However, it is too conservative in its computation of possible event preemptions to be viable in a distributed environment.

¹¹There is a direct correspondence between a state variable in the specification and one in the implementation of the specification. This is dictated by storage management considerations.

3.2.2. Implementation

AIDE implements a simulator as a flavor instance running in a process that maintains a simulated-time-ordered *eventlist* and an associated *global clock* for a given device. At every execution step the simulator removes the event at the head of the eventlist, moves the clock to the specified simulated time, makes the appropriate state change, and invokes the behavior method of the relevant component. Events generated by the behavior of a component are passed back to the simulator, which sorts them into the eventlist to be processed when they get to its head. It is easy to see that this algorithm satisfies the acceptability criterion we defined in Section 3.1.2.

AIDE uses the graphical interface to allow the designer to access the simulator associated with a device. It provides operations to reset, initialize, and run a simulation with or without breakpoints [10].

Current facilities for "observing" a simulation are limited; a general instrumentation interface is under design.

3.3. Distributed Simulation

The motivation for distributed simulation is doing event processing in parallel using multiple machines to gain a reduction in the overall simulation turnaround time as compared to a sequential simulation. Thus, synchronization algorithms for distributed simulation systems seek ways of processing non-preemptable events in parallel. These algorithms must trade off the cost of determining potential event preemptions against the cost of processing the events themselves in minimizing the total execution time of the simulation. Such costs, naturally, depend on various factors, including the target machine environment. Our discussion below assumes a machine environment that consists of small number of fairly powerful machines (e.g., *Symbolics* 3600s) communicating over a shared network (e.g., an *ETHERNET*).

Though there are various classes of synchronization algorithms for a distributed environment [8], we only consider those which distribute control of the simulation to the participating machines, that is, algorithms that are run individually by each machine. Such an approach will avoid creating unnecessary bottlenecks at some central controller.

3.3.1. Partitioning the Problem

Decomposition is not only a powerful tool in design but also in distributed problem-solving. It is therefore natural to consider various ways of partitioning the problem of simulation into subproblems which may be tackled by the participating machines individually. In doing this partitioning we must keep in mind that we would like each machine to operate as autonomously as possible and also that there are costs associated with communicating information between machines which we would like to minimize.

Usually the structure of a device (system) directly reflects its functionality. Given the nature of the design representation, this implies that the subcomponents of the device themselves behave fairly autonomously. This in turn points to the obvious utility of partitioning the simulation problem by assigning to each machine the subproblem of simulating some subset of the components of the device (system). Such a partitioning approach will tend to reduce the gross interactions and shared state between the machines, thus reducing the costs associated with communicating and keeping consistent such information. It will also allow each machine to operate reasonably independently. Furthermore, if the system being modelled *itself* exhibited concurrent activity (a multiprocessor computer system like CARE, for example), then this partitioning scheme may enable the overall simulation of the system to directly exploit the natural parallelism visible in the events that represented the "actual" concurrency. The above are, in fact, basic assumptions of the AIDE distributed simulation approach, as they are of most other distributed simulation schemes [6, 3, 8, 9].

3.3.2. The Basic Execution Model

Before going into detail about the ADE synchronization scheme, we describe here the underlying organization of the distributed simulation system. This particular organizational choice was based on the expectation that the number of components in the distributed simulation will be far greater than the number of machines (thus making an approach that used a machine-level process to simulate each component impractical) and on the likelihood of the consequences of events propagating from any block of the partitioned device to any other block. These assumptions are justified by an analysis of the CARE architectural specification.

The designer initially selects a set of machines to use in the simulation. The system initializes a *simulation server* process on each of these machines; a server is essentially a sequential simulator plus support for synchronization. Thereafter, the designer logically partitions the simulation by assigning participating components to servers. Upon compiling any relevant synchronization information (see Section 3.4.1), the system physically distributes the components as specified in the partition to the relevant server machines. This has the effect that the *lines* that constitute the connection paths between components on different machines now also run between the relevant machines.

The execution of a simulation consists of a sequence of biphasic simulate-synchronize cycles by the servers. During the *simulate* phase of the cycle, a server processes those events that (according to the information it possesses) cannot be preempted by any other pending event in the system. After processing these events, a server enters the *synchronize* phase of the cycle. Each server communicates (an abstraction of) the state of the simulation of its local components to the other servers and then awaits receipt of similar communications. Thereafter, each server interprets these synchronization messages to determine its next set of extant non-preemptable local events.

3.3.3. Conditions for Speedup

In the context of this system organization, we can identify three desirable goals if we are to achieve speedup.

- *Maximum concurrency.* The greatest possible number of servers should be actively processing events during the simulate phase of any system cycle.
- *Maximum ratio of simulation work to synchronization overhead.* The simulation work done by each server during every system cycle should be maximized with respect to the overhead of gathering and communicating synchronization information. This is equivalent to saying that the number of synchronization points should be minimized over the duration of a distributed simulation.
- *Balanced load.* The simulation work done by each of the actively simulating servers in any cycle should be about the same so that the busiest server will not slow down the entire system.

3.3.4. Using the Device Specification in Determining Preemptions

Synchronization based on the simulated times of events alone unnecessarily (and, in most cases, severely) restricts the amount of exploitable concurrency. Few hierarchical models (e.g., CARE, which mixes detailed simulation of inter-processor communication with more abstract simulation of intra-processor computation activities) exhibit much coincidence at the event level. There are very few events at any given simulated time and so there will be very few opportunities for parallel processing in their simulation. The problem lies in assuming that an event with simulated time t could be preempted by any event for simulated time less than t . The functionality of the device being simulated, that is, the device specification, and the behavior requirements (see Section 2.3.3) provide additional information for better estimating potential event preemptions.

Since the preemption relation applies between events, the more information contained within an event is used by the synchronization algorithm, the closer its synchronization activities comes to using the results of the simulation itself and the better its estimation of preemption relationships. We organize the synchronization information that an event carries in terms of the "fields" of an event "record".

1. *Simulated time* is essential in synchronization, as the definition of preemption and the implementation of a behavior specification already suggest. We may make use of the property that two events with the same simulated time are always independent to find inherent parallelism.
2. The *component* is also useful within a partitioning scheme like ours that exploits the structural topology of the device. As a component has a minimum, positive simulated time *delay* between consuming an event and generating one on an output port, and since it is directly *connected* to only some small subset of the other components, an event for that component will have a simulated time "lag" before it could preempt an event on a component more "distant" in terms of connections. This enables a server simulating the latter component to consume in parallel existing events for it up to "lag" simulated time units beyond the event for the original component. Connection information is available in the structural specification of a device and minimum delays may be extracted from component behavior specifications.
3. The *state property* being changed within a component is useful when a behaviorally complex component has a number of internal states that affect its output ports with varying delays. This gives better bounds for "lag" on a per-event basis within such a component, thereby giving a better overall approximation of possible preemptions. Such information can be determined as for the component itself.

Much of the above synchronization information can be efficiently compiled before the actual execution of a partitioned simulation. However, some of it must still be computed dynamically by the machines, communicated between them and, finally, used by them. This results in *overhead* costs that could undermine the speedup gains realized by the increased opportunities for parallelism.

3.4. The AIDE Distributed Simulator

In this section we examine in some detail the synchronization algorithm used by AIDE. The algorithm reflects a particular choice in the complexity of the synchronization information that is maintained, communicated and interpreted by the AIDE simulation servers. We conclude with some performance data for a simulated multiprocessor device and provide an analysis of this data with respect to speedup.

Each simulation server, as mentioned above, is essentially a sequential simulator (operating on a piece of the device) with support for system-wide synchronization. Each server maintains an eventlist and a local clock whereby local *non-preemptable* events are processed as in a serial simulator (this is not quite true, see Section 3.4.1). It is thus convenient to abstract synchronization information into information regarding the simulated times of possible inter-server preemptive event arrivals and departures. This is precisely what AIDE does.

3.4.1. Static Synchronization Information

Partitioning in AIDE begins by the user selecting a partitioning level in the hierarchy of components that constitute the device. Every component at this level is considered to be a *logical process* or *lp*. This means that, unless the simulator is informed otherwise, simulated time at any point in the execution of the simulation will be consistent within the *lp* and all of its subcomponents. However, different *lps* may have different simulated times, even on the same machine.

Thereafter, the user logically partitions the set of lps into server blocks. Each block corresponds to a server (machine) that will simulate its member lps during the execution of the distributed simulation. This partitioning is currently done interactively via the graphics-based interface.

The synchronization information now compiled is at two levels : the intra-server level and the inter-server level. In both cases, the information is static during simulation.

At the *intra-server* level, the following quantities are computed and cached within a component. They use a required, predeclared, positive minimum event delay that is associated with each top-level behavioral specification by the designer.

1. The window-in (WI) of a component is the minimum simulated time delay for an event received on any "border" input port (in the local partition block) to have a consequence on a port or state of the component.¹² Components with border input ports, or border consumers, have $WI=0$. WI quantifies the equivalence of a topologically "inland" lp to its "closest" border consumer lp in the same partition block, in terms of the propagation delay experienced by preemptive events arriving at the border lp during simulation. It is computed by simple path analysis within each block of the partition, in conjunction with minimum component delays.

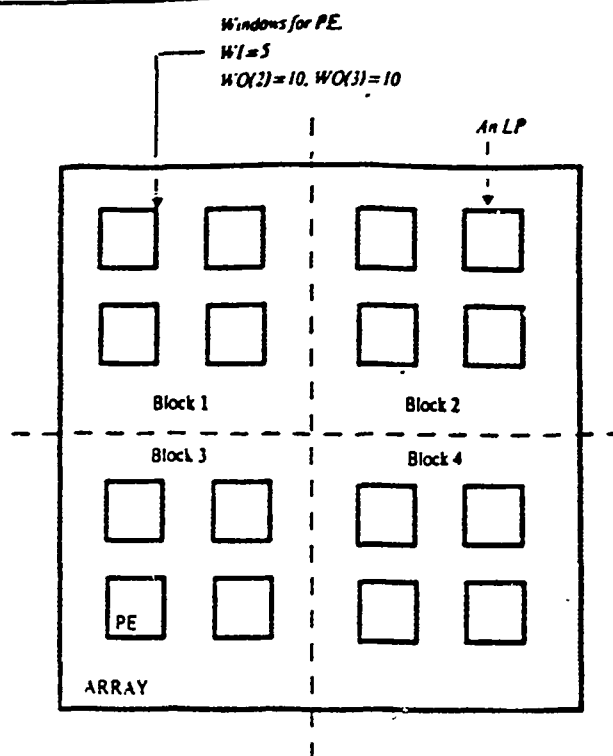
During execution, the effective simulated time (EST) of an event is the simulated time of the event minus the WI of the component concerned. Events at a server are sorted and consumed in EST order in an effort to reduce synchronization points by increasing the number of events that can be processed in a given simulation phase. By this technique, local lps can often possess different simulated times and yet be correctly simulated in the same cycle. This also enlarges the work to overhead ratio of an execution cycle.

2. The window-out (WO) of a component is the minimum simulated time delay for any event that is consumed by the component to have a consequence on each "downstream neighbor" block (of the local block). The WO of a component with border output ports (a border producer) to the directly downstream neighbor block(s) is the minimum event delay associated with the component. WO quantifies the potential preemptive impact of an unprocessed event at a component on each downstream neighbor block of that component's block. We note that the utility of WO is predicated on the assumption that the device will be partitioned in such a manner as to cluster connected components together on a server and that there will be a small number of downstream neighbor blocks for each block of the partition. We also note that, unlike WI which is based on an lp, WO is calculated for each individual participating component since it does not affect time consistency within an lp. The WO delays are calculated using path analysis within each block of the partition, along with the minimum delays on components.

During execution, the WO record of the relevant component is stored with each produced event. At the beginning of a synchronize phase, the WOs on all extant events are used by a server to calculate the earliest times any unprocessed local event could affect each of its downstream neighbor servers. Such neighbor impact times are communicated in a synchronization message to the other servers. Each recipient server can abstract these neighbor impact times into impact times at every server (including itself) by using the inter-server delay table (see below).

¹²A border port is one that is connected with a port of a component in some other block of the logical partition. Its owning component is a border component.

Windows can be overridden during the simulation (see Section 3.4.2).



Array of PEs topologically partitioned into 4 server blocks.
Lines and Ports not shown; PE delays = 5 units.

Figure 3-1: Example of a Partitioned Device

At the *inter-server* level, the WO quantities are abstracted into a table of inter-block event-consequence propagation delays. The system conducts shortest-path analyses on the blocks of the partition and constructs a table that associates with each block and each of its downstream neighbor blocks the minimum event impact delay to every other block. Hence, when a server says that it has a local event that could affect a downstream neighbor at a certain simulated time, every server can use this table to determine the minimum additional delay before the consequences of that event could traverse the downstream neighbor and ultimately have a local preemptive impact.

3.4.2. Dynamic Synchronization Information

As mentioned in Section 3.3.4, complex components often exhibit varying consequence delays for events on different state properties. This is particularly true of an abstract model like CARE. Also, sometimes when an event is generated for some future simulated time, the functionality of the component in question may imply that it can be consumed "eagerly" or with a higher WI, and thus, lower EST (e.g., when a subcomponent of an lp does not require the time consistency provided by default). ADE allows the designer to convey this sort of information to the simulator by allowing metabehavioral declarations for each component class. These declarations take the form of propositions that give scheduling hints to the local server in terms of better bounds on the WI and WO of the component for events on specific state properties. Such bounds are computed relative to the component's I/O ports at event-generation time and passed to the simulator to be generalized to the local partition block.

ADE provides the `metabehavior` form to declare the metabehavior rules and the minimum event

propagation delay of a component class. It has a compiler to transform these rules into more efficient procedural code. Details may be found in [10].

3.4.3. Synchronization Algorithm

Once the static synchronization information has been compiled, the device is distributed as specified in the partition. Components in each block of the logical partition are physically migrated to the corresponding server machine (see Appendix I). The simulation is initialized. Figure 3-2 shows the physical system organization of a simulation that has been distributed over 4 machines. Note that the bidirectional network streams between server machines carry events (over the lines of the distributed device) as well as synchronization messages.

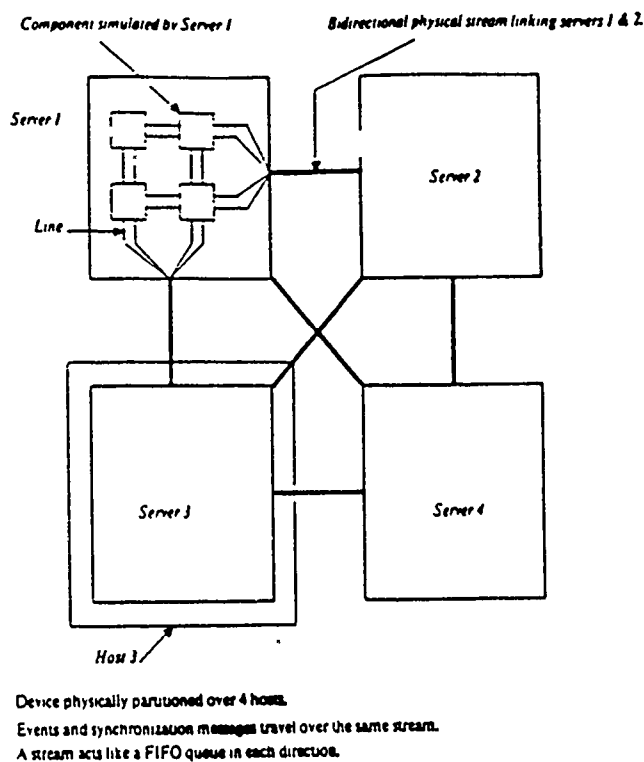


Figure 3-2: Typical Organization of a Distributed Simulation

As mentioned above, each simulation server is a sequential simulator with additional capabilities for synchronization. These capabilities include data-structures and procedures for generating, transmitting, receiving and interpreting *synchronization messages*.

A synchronization message sent by a server consists of :

1. The local server's unique identifier. For efficiency, this is an integer that is used as an index into various tables.
2. The EST of the event at the head of the local server's eventlist at the instant that the synchronization message was sent.
3. An alist of the local server's downstream neighbor servers and the lowest neighbor impact

times (see 3.4.1) of all the local server's unprocessed events with respect to each of these. This alist describes the preemptions that could happen in the next simulate phase.

4. An alist associating with each of the local server's downstream neighbor servers the earliest simulated time of any events that were generated in the just-completed simulate phase that were sent (over component lines) to that neighbor server. This is the local server's **preemption alist** and it describes the preemptions that actually happened in the previous simulate phase.
- **Generation.** Synchronization messages are generated in the server process. EST and neighbor impact times are obtained by scanning the eventlist. The preemption alist is obtained via a table that is updated by the ports that forwarded events to an input port on some downstream neighbor server in the preceding simulate phase. This table contains the earliest simulated times of such events, if they were generated. It is reset at the end of a synchronize phase and updated during a simulate phase.
- **Transmission.** Synchronization message transmission is done in the server process. The server gains access to the stream (see Figure 3-2) to each server and sends (a copy of) the message on that stream. Since events are generated in the server process only, and since a stream acts as a FIFO queue, we are guaranteed that synchronization messages will arrive at the remote server after any transmitted preemptive events and after all previous synchronization messages.
- **Reception.** All stream reception is done asynchronously in a single background process (see Appendix I). As preemptive events are received over a network stream, they are atomically entered into the local server's eventlist. As synchronization messages are received over a stream, they are atomically queued in a FIFO queue reserved for the sending server. The server process polls this queue to determine whether a synchronization message has been received from the sending server.
- **Interpretation.** Interpretation is done in the server process. Each server maintains 2 tables, a **next-EST-table** and a **neighbor-impact-table**, each of which has an entry for each server in the system. After all synchronization messages in a given cycle have been interpreted (as described below), the next-EST-table contains the correct lowest EST of each server in the system and the impact-table contains the correct lowest neighbor impact times (see Section 3.4.1) for each server from any of its upstream neighbor servers.

During execution, an AIDE simulation server runs the usual cycle of two phases.

1. **Synchronize.** If the server was active (processed some local events) in the preceding simulate phase, it generates a synchronization message as described above. A copy of this message is transmitted to each server in the system.

Each server now uses the next-EST-table that was computed in the *preceding* synchronize phase to determine which servers were active in the preceding simulate phase. This determination is analogous to that done to find out whether the local server was active (see below). It waits to receive a synchronization message from each of these servers. Once a server determines that synchronization messages have been received from all the active servers, it interprets these messages.

The next-EST-table is filled in first. The EST reported in each synchronization message is initially entered for the corresponding server. Then the preemption alists are merged into the

table entries -- an EST is replaced if a preemptive event with a lower EST was forwarded to a server from one of its upstream neighbor servers.¹³ Then the neighbor-impact-table is updated. Initially, all reported neighbor impact times from all non-preempted servers (see previous paragraph) are minimized into their downstream neighbor servers' corresponding impact-table entry. The neighbor impact times for each preempted server are recalculated (using the compiled inter-server delay table) and merged into the impact-table.

At this point the next-EST-table and neighbor-impact-table correctly reflect the system state. Now the local server extrapolates the neighbor impact times to determine whether it will be actively simulating during the next simulate phase. This is done by using the inter-server delay table to calculate the local preemption time (PT), which is the minimum impact time of a preemptive event arrival from any other server via a path through any of its downstream neighbors during the next simulate phase. If the local EST (taken from the next-EST-table and thus correct) is less than PT, then the server will be active; otherwise, it will be inactive. For efficiency, similar activity calculations for the other servers in the system are deferred until the local server has to wait for them to report in the next synchronize phase (see above).

2. **Simulate.** If the server determines that it is active in this phase, it processes all the existing local events that have EST less than PT (making use of WI). Events from upstream servers may be asynchronously received for border input ports but they will be for simulated times not less than PT. Similarly, events on local border output ports may be transmitted asynchronously to remote input ports. As local events are generated, they are sorted by EST into the eventlist; they may be safely processed in the current phase if their EST is less than PT.

During the execution of a distributed simulation, the user may interactively issue commands to the servers to break, continue, or reset the simulation. The system currently provides no instrumentation capabilities aside from a general text trace facility.

3.4.4. Acceptability of the Algorithm

To show that this algorithm is acceptable, we first note that the simulation activity of a server during any simulate phase does not violate the acceptability condition if we assume that the minimum event delays and, hence, WI calculations, are correct. This can be seen in the fact that the server essentially runs a sequential simulation in this phase, processing all existing events that cannot be preempted by an external arrival at a border input port (which will have a simulated time of at least PT) during that phase.

Secondly, the (single) network communication stream between any two servers behaves as a virtual queue for preemptive events and synchronization messages. Therefore, preemptive events (from an upstream neighbor) show up in a server's eventlist *before* a synchronization message denoting the completion of the current simulate phase at the upstream server is received. Therefore, local simulation state is consistent before a server performs the activity calculations for the next simulate phase.

Thirdly, incorrect information regarding minimum ESTs (caused by the interval between sending a synchronization message and receiving all other synchronization messages, during which an event with an earlier local EST could be received from an upstream neighbor server) is overcome by having each server

¹³The preempted server may have generated its synchronization message *before* the arrival of the preemptive event. Hence, the local EST it reported may be inaccurate. This could happen if the preemptive event had a lower simulated time and, hence, EST (because the component of the preemptive event will have WI=0), than the event that was previously at the head of the server's eventlist.

include in a synchronization message the minimum times of remote events generated for every downstream neighbor server during the preceding simulation phase. Hence, once all the required synchronization messages from active servers have been merged in a synchronize phase, a server's next-EST-table correctly reflects the state of the entire system. If component event delays are correct, then so are the WO calculations and so is the precompiled inter-server delay table; therefore, a server's neighbor-impact-table at this point will accurately reflect the earliest possible preemptive arrivals at each neighbor, and the PT and activity calculations (which extrapolate neighbor delays using the delay table) will also be correct.

A formal proof of acceptability can be constructed by following the methodology given in [3].

3.4.5. Deadlock Avoidance

Peacock et al have shown that a cyclic path of connected zero-delay components is a necessary condition for deadlock to occur in a distributed simulation [8]. Since AIDE requires that components have a minimum, positive delay, and since a server simulates at least one component, this condition can never be satisfied in a distributed AIDE simulation. Hence, deadlock is avoided.

3.4.6. A Probabilistic CARE Model

To evaluate the performance of the AIDE algorithm, we use a probabilistic model of a multiprocessor based on CARE. The characteristics of this model were induced from CARE's event history in a serial simulation. As mentioned earlier, CARE exhibits clusters of "communication" events (representing packet routing between nodes) that are localized in simulated time as well as over the processor grid. These are intermixed with slower "computation" events (representing processing activity within a node) that have larger, more varying simulated time periods. In using a probabilistic model, we bypass many of the additional software issues involved in distributing CARE programs while still retaining information that allows us to predict the performance of a distributed simulation of the architecture.

The device to be simulated consists of a quad-connected array of processing elements (PEs). Each PE does both computation work and local message routing in the array. The quantity and real and simulated time durations of the events consumed by a PE are controllable through the use of parameterized probability distributions to represent each type of activity. Typical sizes of the array are 64 and 256 PEs.

3.4.7. Performance Analysis

Table 3-1 presents data collected for distributed simulations of two models. The first five experiments were done with a 256-PE probabilistic CARE model on 1, 2 and 4 machines. This model was parameterized to exhibit different computational characteristics by varying the real-time to process events (F=fast, S=slow), the number of initial events (H=heavily loaded) and their staggering in simulated time (D=dispersed), and the regularity of event consequences (R=more regular, N=normal, CARE-like).

The sixth experiment used a different behavioral characterization of the CARE structural model in which intra-PE "computation" was specified in detail while inter-PE "communication" was more abstractly specified. The array was heavily loaded, with initial events at each PE. The purpose of this particular experiment was to test AIDE's performance on a model that exhibited the desirable characteristics for speedup delineated in Section 3.3.3.

In each experiment, *speedup* was measured by comparing the event processing rate (in events processed per elapsed second) achieved by the distributed system to that attained in a serial simulation of the device. *Concurrency* was measured by sampling the number of servers that actively simulated in a system cycle (the sampling was done every cycle).

The data is presented in terms of the time breakdown at a typical server both in every cycle (Table A) and in every *active* local cycle (Table B). In other words, Table A averages a server's performance over

all system cycles (irrespective of concurrency), and Table B averages a server's performance over those cycles in which it was actively simulating. The latter gives a better idea of system performance per cycle since it takes into account the effects of concurrency.

The important observation is that, for the usual CARE model, only experiment I achieved any performance improvement over the serial case (225/200). Furthermore, this speedup was at best marginal, although average concurrency was nearly optimal (3.9/4). However, experiment VI with the "good" model demonstrated an excellent performance improvement; the speedup factor was 1.73 out of a best-case 2. In the other experiments, the ratio of simulation work done to synchronization overhead was simply too low, resulting in performance losses, so that the system could not even match the serial event processing rate.

Another observation is that serious limits on server concurrency appear to be unavoidable in a model with the structural and behavioral characteristics of CARE. This is because (1) the model attempts to model inter-PE behavior (communication) in much more detail than intra-PE activities (computation), (2) packet routing done by the PEs is based on the dynamic network load in the array, and (3) the PEs are *closely-coupled* in that each PE can route a packet to any other PE in the array with a small number of fast inter-PE routing hops. It is typical, therefore, that the appearance of a communication event at any PE within a server severely limits the "looseness window" or degree of independence of the other servers, because it could potentially have a consequence on any PE in the array in the very near simulated time future. This ultimately curtails the concurrency (and therefore the speedup) of the system because of the similar consequent communication events that result as the CARE packet moves through the inter-PE lines. Additionally, when pending PE computation events are consumed, they tend to have communication consequences for simulated times far beyond those calculated using the inter-server event propagation delays on packets being routed in the array. Comparing experiments I and II demonstrates this; in experiment I, the "regular" model's event pattern was such that the consequences of computation events were generated with a delay that was close enough to the inter-server windows to raise concurrency to 3.9/4 (versus 2.2/4 in experiment II with the "normal" behavior model).

The close coupling of PEs also decreases speedup because of all the associated synchronization overhead (communication and serialization) that emerges from the resulting close coupling of the simulation servers. A large part of a typical CARE event set consists of the serializing communication events, well-staggered in simulated time and well-distributed over the array, and the dynamic routing scheme of CARE makes it impossible to bring any better knowledge to bear on the preemptive bounds of such events to help the servers loosen their event-processing windows. The "good" model, on the other hand, manages to sufficiently decouple the PEs at the simulation level to attain a marked decrease in the number of synchronization points for the distributed simulation, with corresponding performance gains.

One final reason for the poor performance of the system is that it often is the case that even if concurrency is optimal there are actually only very few events to process at each server. The data shows that it is *much* cheaper to process an event (of the order of 1 millisecond) than to send and receive a synchronization message (of the order of 100 milliseconds). Hence, in experiments II and III we see that lower concurrency leads to a better overall event-processing rate.

Where does the time go during synchronization? Table 3-1 shows that sending a synchronization message takes about 20 milliseconds per remote server in *local* processing overhead at the sending server (see row Local Stream). This overhead consists of byte-coding the LISP data, packet allocation, and the protocol-level overhead associated with stream reliability (e.g., packet sequencing, acknowledgements and retransmissions) and network interface access. There is a greater local processing cost at the receiving server; besides the costs mentioned above, packet reception costs include process-switching for the scheduler and the network receiver processes and various protocol-level checks to ensure the integrity of the data contained in a received packet. Unfortunately, an active server has to send a synchronization message to *every* other server during a synchronization phase, so this cost grows with concurrency and with the number of machines. However, only active servers contribute to this cost, so we have made some gains over the naive $O(N^2)$ approach, and high concurrency is quite rare in practice.

Experiment	I	II	III	IV	V	VI*
Total Server Machines	4	4	2	2	2	2
CARE Model Type	R,F	N,S,H	N,S,H	N,D	N,H	'Good'
Avg Concurrency	3.9	2.2	1.5	1.3	1.6	2
Avg Event Rate (e/s)	200-250	60-70	100	124	140-150	551
Serial Event Rate (e/s)	200+	120	130	180	180?	318
A. Server performance per SYSTEM cycle						
SYNCHRONIZE (Ms)	343	188	129	83	103	1312
Send Sync (Ms)	66	38	15.7	12.15	15.84	35.5
Compute (Ms)	3	1.5	1.85	1.15	1.84	19.9
Local Stream (Ms)	63.3	36.5	13.83	11	14	16
Receive Sync (Ms)	275	150	111	70	87	1276
Compute (Ms)	2.5	2.4	0.04	0.05	0.04	<1
Wait & Stream (Ms)	273	147	111	70	87	1276
Interpret Sync (Ms)	2	2	0.54	0.53	0.53	1
SIMULATE (Ms)	161	52	141	58	108	11853
B. Server performance per local ACTIVE cycle						
SYNCHRONIZE (Ms)	342	218	130	88	106	1312
Send Sync (Ms)	67	68	20	18	19	35.5
Per Stream (Ms)	22	22	20	18	19	35.5
SIMULATE (Ms)	163	88	183	90	135	11853
Folding Simsteps (#)	1.85	2	3.9	3.7	3.6	57.9
Remote Events (#)	1.2	0.28	0.21	0.13	0.28	5.8
Remote Events (Ms)	26	6	4	2	5	112

Notes

- Table A breaks down a typical server's activity in a system cycle as time spent in simulation and synchronization. Synchronization consists of sending, receiving, and interpreting sync messages. The first two of these have a computation cost and a communication cost.
 - Table B reflects the fact that only *active* servers in a system cycle actually send synchronization messages and simulate. However, as every server must receive sync messages, we do not repeat that data here.
 - *Local Stream* denotes local overhead in getting a message to a network stream.
 - *Wait & Stream* denotes the average wait until the last sync message for a system cycle is enqueued at the recipient server. It does not distinguish load imbalance from message latency.
- * The 'good' model is discussed in the text.

Table 3-1: Performance Data for 256 PE Probabilistic CARE Model

Based on the above analysis, we may assume that the average latency of a synchronization message is at least 40-50 milliseconds per remote server. The Wait on Stream numbers in Table 3-1. measure the average time a server (whether active or not) spent per cycle in waiting for the last synchronization message to be locally received. As such, this data cannot distinguish point-to-point message latency from cases in which synchronization messages were simply transmitted late due to the transmitting server being busy-simulating, or, conversely, cases in which synchronization messages from less busy servers were already queued at a busily simulating server before it started the wait. Load imbalance is the most probable reason that the servers in experiment VI had an average synchronization wait of 1276 milliseconds even though concurrency was optimal; message latency cannot be more than a minor part of that wait!

3.4.8. Performance Characterisation

In summarizing our performance results, we return to the conditions for speedup put forth in Section 3.3.3 and characterize the AIDE algorithm with probabilistic CARE in terms of these.

- Maximum concurrency. AIDE tries to maximize concurrency in the simulation by precomputing good static lower-bound path delays, allowing the user to specify better dynamic bounds, and maintaining and using WO to each neighbor server in determining better path delays for inter-server preemptions. The improvement in concurrency over an early implementation that simply did the first of the above (and never achieved more than about $1.1/N$ concurrency) has been impressive. Furthermore, the computational overhead introduced by maintaining and using this information has been completely swamped by the communication cost of exchanging it, except in experiment VI, where both costs were comparable.¹⁴ However, contrasting experiment VI with the others shows how a suitable model can make good use of this capability. A CARE-like model appears unsuitable by virtue of its close coupling and staggered, irregular event pattern.
- Maximum ratio of simulation work to synchronization overhead per active step. The WI and WO mechanisms serve to lengthen the simulation phase at a server. However, these mechanisms are only useful insofar as the structural and behavioral characteristics of the model admit; generally, the features that adversely affect concurrency also tend to decrease the ratio of simulation work to synchronization overhead. The beneficial effects of the AIDE mechanisms can be estimated by the number of simulation steps that occur per active step in Table 3-1. Experiment VI with the "good" model averaged nearly 60 simulation steps per synchronization at each server, and the simulation work per active step was an order of magnitude greater than the synchronization work. Not visible in the table is the observation that WI often results in events in the very near future being simulated, by virtue of their EST, in the current simulation step and hence (some of) their consequences being effectively hidden from synchronization. The ratio of useful work to overhead decreases with more machines since the communication cost of synchronization usually increases and the average simulation work per server decreases. On the other hand, concurrency, and hence the potential for speedup, tends to increase with the number of machines (to a certain point). This creates a difficult trade-off in determining the most effective number of machines to use in the simulation of a given device.
- Load balance. The effect of load-balance on speedup is perhaps the hardest facet of the system to quantify. There is no question that load balance is as much a function of the behavioral characteristics of the model (regular events vs. more random events, the number of

¹⁴This was due to the large number of events that had to be searched in determining neighbor impact times.

events processed at each step and the work involved in processing them) as of the partition. The effects of this are hard to measure in the current implementation of AIDE because it does not have sufficiently detailed measurement capabilities. It would be interesting to determine whether there is any correlation between the load-balance of an application atop CARE and the load-balance of a distributed simulation of the architecture running this application. Other factors affecting load-balance include the activity of the incremental garbage collector at a server, relative server machine speeds, and network activity.

4. Conclusions and Research Questions

In AIDE we have created an environment that is capable of capturing and validating interesting designs. The system can also serve as a testbed for investigating the performance of distributed simulations of such designs. We have carried out such an investigation for a probabilistic model of a multiprocessor system that is based on the CARE machine architecture.

4.1. Distributed Simulation

AIDE uses its knowledge of the device being simulated to increase both the inter-server concurrency (coarse-grain) as well as the ratio of simulation work done to synchronization overhead (akin to fine-grained concurrency) achieved in a distributed simulation. Empirical evidence indicates that such knowledge is essential if distributed simulation is to realize any performance gains, and that the mechanisms through which the AIDE system exploits this knowledge are effective. However, some models are inherently limited in the amount of concurrency they allow. Tightly-coupled models that do mixed-mode simulation (for example, CARE) tend not to admit much coarse-grained concurrency; synchronous models (for example, logic circuits) and loosely-coupled, mixed-mode models tend to admit much more. Since the coupling at the model level has a direct counterpart in the coupling at the server level (by virtue of the topological partitioning scheme), this translates into increased synchronization overhead that usually results in performance degradation.

In addition to concurrency, the simulation work done between synchronize phases in a distributed simulation must be substantial, relative to the overhead of synchronization, if any performance gains are to be realized. The indications from our experiments with the AIDE system are that, for speedup to be realized, either

- the processing of a typical event (that is, a behavior invocation) must involve substantial computation, or,
- a large number of events must be processed between synchronizations, or,
- communication costs must be reduced.

We feel that in the simulation of CARE-like devices that use closely-coupled components and mix detailed simulation of inter-component interaction with more abstract simulation of intra-component function, achieving the first two goals is unlikely to yield satisfactory performance improvements. The prognosis is better for the simulation of very large CARE-like devices on a small number of machines or for the simulation of more loosely-coupled or synchronous devices. However, the third condition seems desirable in all cases, and it is probably achievable through software or hardware optimizations. This promises substantial speedup gains for suitable models.

Lastly, the AIDE synchronization algorithm is not sufficiently well-suited for closely-coupled models that exhibit a behavioral activity pattern in which events are well-scattered over the structure of the model. AIDE's structural partitioning scheme, in conjunction with the relatively close behavioral coupling between different structural "pieces" of the partitioned device, directly translates into load imbalances during the

distributed simulation of such a device. The end-effect of this is the serialization of the distributed system, often resulting in performance losses. The effects of imbalance tend to diminish in loosely-coupled devices and in those that exhibit relatively synchronous events in a most parts of their structure.

4.2. Open Questions

There are some obvious issues that need to be resolved with respect to the distributed simulation algorithm in ADE. The first task is to identify and reduce the communication overhead seen in synchronization as this has the potential to make a vast difference in performance. Another task is to attempt a reduction in the total number of synchronization messages that are sent during a simulation, thus improving the simulation/synchronization cost ratio. Perhaps some type of multicast could be developed, thus decoupling the cost of synchronization from the total number of machines, or the basic synchronization algorithm could be modified so that an active server need not inform every other server of its local state. Also, if the costs of communication could be substantially reduced, it would be worth investigating whether the lock-step synchronize-simulate system cycle could be improved upon to reduce the effects of load imbalance on performance.

Finally, it appears to be a challenging problem to actually implement the CARE specification in ADE and execute distributed simulations of the concurrent machine "running" application programs. We expect that this will raise interesting problems in the areas of distributed debugging, distributed instrumentation, and data consistency over multiple Lisp environments. We hope that improvements in the performance of the ADE synchronization algorithm will make such an effort worthwhile.

I. The Distributed Object Subsystem

In this appendix, we briefly summarize the extensions to the ZETALISP programming environment that support the distributed environment of AIDE. The underlying idea is to emulate a single address-space over multiple address-spaces by implementing *remote-addresses*. This implementation is done through the cooperation of three types of entities -- a *transaction stream manager*, an *interned object*, and a *remote object*. Each of these is implemented via the FLAVOR system.

An interned object is the only Lisp object that is required to exist as a single incarnation in this emulated single-address-space. Its representation on a remote machine that needs to "point" to it takes the form of instance of a remote object on that remote machine. Such a remote object essentially contains a local pointer to the transaction stream (called its *access-stream*) that links the local host to the host that contains the real interned object, and a unique ID within the context of that stream that resolves to its real incarnation on the other side. Such a bidirectional transaction stream manager exists between every pair of hosts.

By requiring that all (data and function) accesses of interned objects be through the form of *messages* sent to (a local pointer to) that object, we can have a remote object can trap all accesses (messages) at the local host and forward these to its real incarnation at its residence host. There the appropriate *method* gets invoked, either in a default background process if the call is for effect, or in a new process if it is for value. When a message is sent down a transaction stream for value the sending process blocks until the result is received by it (in an emulated *value cell*). When a message is sent for effect (for example, an event from an output port to a remote input port during distributed simulation), the local sending process simply continues as normal. Whether a message is sent for value or effect is currently programmer-specified. References to non-interned objects always result in an isomorphic copy of that object being transferred across the transaction stream.

Such remote method invocations are rendered fairly efficient by using an extension of the Lisp-machine BIN-file data transfer protocol. The generality of the transaction stream interface also allows the migration of partial networks of instances to various machines by copying them to the destination machine and replacing all local pointers to them by *forwarding pointers* [13] to newly-created remote objects (as is done during distribution of the components of a design), the shadowing of state and messages by remote objects (as is done for the *name* slot of all AIDE components), remote procedure and function calls (by transferring a function specifier or function along with locally evaluated arguments and funcalling it remotely). The system provides various operators to carry out such operations, detailed in [10].

Transaction streams are built on reliable byte-streams. We use a system of locks to guarantee the consistency of a logical message transfer along the stream, since the stream may be multiplexed between many processes and interned objects.

The distributed object subsystem has proven its utility time and time again during the development of AIDE. Its basic mechanisms currently implement functionality that ranges from user control of the simulation servers to cooperative event transformation between components to instrumentation. As an illustration of its convenience and power, we report that an (admittedly naive) implementation of a remote graphics extension took roughly twenty lines of code to specify and about an hour to debug. AIDE's graphics interface does not now distinguish between components that are remote and those that are local.

References

- [1] Arvind, Michael L. Dertouzos, and Robert A. Iannucci.
A Multiprocessor Emulation Facility.
Technical Report 302, Laboratory for Computer Science, Massachusetts Institute of Technology,
October, 1983.
- [2] Harold Brown, Christopher Tong, and Gordon Foyster.
Palladio: An Exploratory Environment for Circuit Design.
Computer 16(12):41-58, December, 1983.
- [3] Randall E. Bryant.
Simulation of Packet Communication Architecture Systems.
Technical Report MIT/LCS/TR-188, Laboratory for Computer Science, Massachusetts Institute of
Technology, November, 1977.
- [4] Bruce Delagi and Jerry Yan.
CARE User Manual.
HPP Report, Stanford University, Department of Computer Science, 1986.
[in preparation].
- [5] Gordon Foyster.
HELIOS User's Manual.
HPP Report HPP 84-34, Stanford University, Department of Computer Science, August, 1984.
- [6] Jay Misra.
Distributed Simulation.
Tutorial notes, ICDCS, IEEE Computer Society.
- [7] A. R. Newton and A. L. Sangiovanni-Vincentelli.
Computer-Aided Design for VLSI Circuits.
Computer 19(4):39-60, April, 1986.
- [8] J. K. Peacock, J. W. Wong, and E. Manning.
Distributed Simulation using a Network of Processors.
Computer Networks 3(1):44-56, February, 1979.
- [9] Charles W. Rose, Greg M. Ordy, and Frederic I. Parke.
N.mPc: A Retrospective.
In *20th Design Automation Conference*, pages 497-505. IEEE, 1983.
Paper 32.1.
- [10] Nakul P. Saraiya.
AIDE User Manual.
HPP Memo, Stanford University, Department of Computer Science, 1986.
[in preparation].
- [11] Narinder Singh.
MARS: A Multiple Abstraction Rule-Based Simulator.
HPP Memo HPP 83-43, Stanford University, Department of Computer Science, December, 1983.
- [12] Narinder Singh.
Corona: A Language for Describing Designs.
HPP Report HPP 84-37, Stanford University, Department of Computer Science, September, 1984.
- [13] *LISP Machine Documentation Set*
Symbolics, Inc., 1985.

A Shared Memory Lisp Package for CARE

Nakul P. Saraiya

KNOWLEDGE SYSTEMS LABORATORY
Department of Computer Science
Stanford University
Stanford, CA 94305

WORKING PAPER

This work was supported by DARPA under RADC Contract F30602-85-C-0012, by NASA Ames Contract NCC 2-220-S1, by Boeing Contract W266875 and by Digital Equipment Corporation.

A Shared Memory Lisp Package for CARE *

Nakul P. Saraiya

January 10, 1989

1 Architectural Model

In [2], we described the programmer's interface to the CARE architectural simulation system[1]. The interface, called LAMINA, provides support for driving CARE models via application programs written in functional, object-oriented, or shared variable styles. In this paper we document a shared memory Lisp package, called QL, that is built upon the LAMINA shared variable model. QL is based on concepts from Multilisp[4] and QLAMBDA[3]. Like them, it provides constructs for expressing parallel computations that augment an underlying serial Lisp system. QL uses Zetalisp[5] as its base language.

The remainder of this section briefly reviews and extends the basic LAMINA shared variable model. Section 2 describes the QL language constructs and section 3 describes its instrumentation facilities. We assume that the reader is familiar with LAMINA as described in [2].

1.1 Shared Memory in CARE

As shown in figure 1, a shared memory CARE model consists of a number of *sites*, each of which behaves either as a processing site or as a memory site. A processing site has processor (*i.e.*, *evaluator* and *operator* [1]) resources (P) and some amount of local memory (m). A memory site has a memory controller (MC), and some amount of shared memory (M). The sites typically communicate via a point-to-point network or a system of busses.

Shared memory cells are constructed from *streams* encapsulated in *remote addresses* [2]. Cells are allocated on memory sites, and processors access them only through memory controllers. Cells hold

*This work was supported by DARPA Contract F30602-85-C-0012, by NASA Ames Contract NCC 2-220-S1, by Boeing Contract W266875, and by Digital Equipment Corporation.

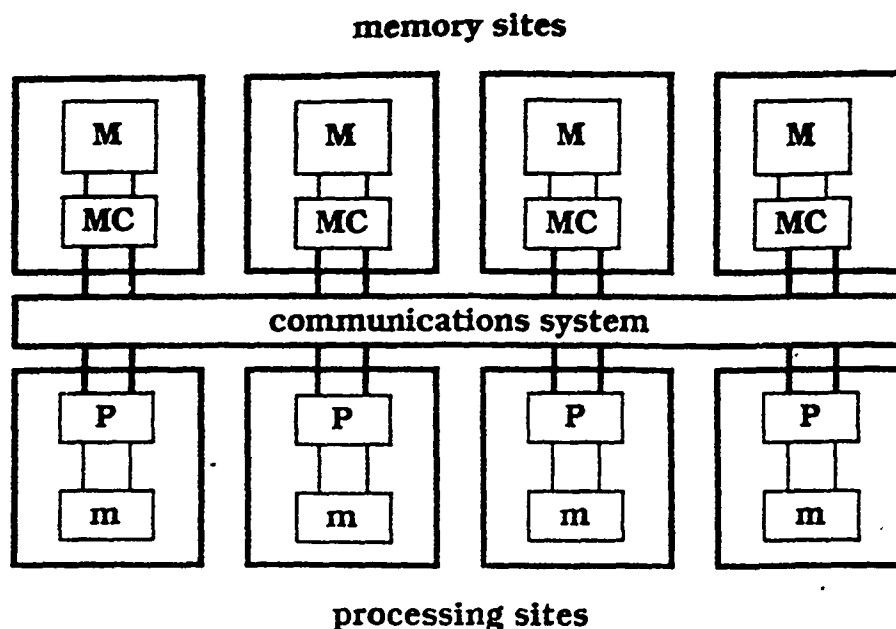


Figure 1: Processing and Memory Sites

a single value at any time. They may be aggregated into data structures such as lists and arrays. LAMINA provides constructs for allocating, reading and writing such data structures as part of its shared variable interface.

Address Spaces

As shown in figure 2, the address space of a LAMINA application is partitioned into **local**, **dynamic** and **static** spaces.

Local space is distributed among the processing sites of the system, *i.e.*, in the local memories of the processors. In LAMINA, data structures created in one processor's local space cannot be directly accessed by another processor. This is further restricted in QL in that local data structures can *only* be accessed by the local processor.¹ They must be copied into dynamic space before other processors can access them; this is discussed further in section 1.2. Data structures in local space are manipulated through the usual Lisp functions executed at the local processor.

¹This is because the QL shared memory model does not currently admit globally accessible memory controllers at processing sites. Such controllers are only available at memory sites. This also implies that structures encapsulated by the **reference** primitive of LAMINA cannot be manipulated with the shared variable operations.

1.2 Extensions to the Shared Memory Model

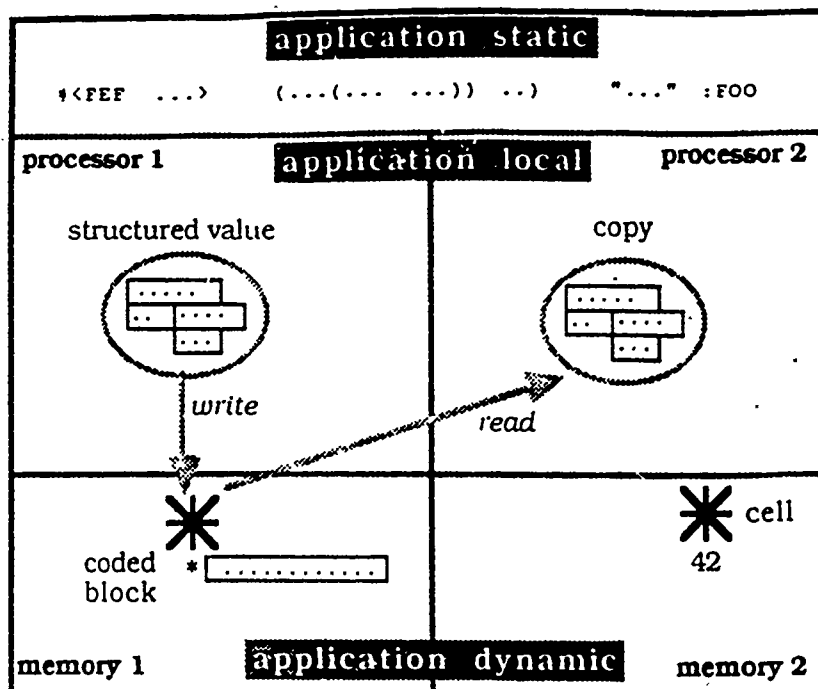


Figure 2: Local, Dynamic and Static Addresses

Dynamic space is distributed among the memory sites of the system.² All sharable data structures exist in dynamic space. They may be manipulated by the LAMINA shared variable operations from any processing site.

Finally, "immutable" objects such as compiled functions and keyword symbols reside in static space. We assume that such objects are available at each processing site (corresponding to a 100% cache hit rate).

1.2 Extensions to the Shared Memory Model

In this section, we describe some extensions to the LAMINA shared memory model that go beyond allocating, reading and writing shared cells as described in [2].

²The exceptions are futures and shared-queues, which are created in dynamic space at processing sites and are managed by the operators at those sites. They can be referenced from all processors.

Coded Blocks

Structures created in local space can be manipulated very efficiently by the maintaining processor because there is no need to traverse the communication facility on every access. However, as mentioned earlier, structures created in the local space of one processor cannot be shared with other processors. They must be first copied into dynamic space. For convenience, we provide an automatic facility to accomplish this that is similar to the means used to communicate values in object-oriented LAMINA.

Whenever a processor attempts to store a pointer to a local structure into dynamic space, the operator at the processing site recursively traverses the structure and copies it into a form that includes only *static references* (to objects in static space), *dynamic references* (to cells and structures composed of them in dynamic space), *internal references* (to subcomponents of the structure value in local space), and *self-referentials* (i.e., "immediate" objects like fixnums and small floating-point numbers). This is then allocated as a contiguous **coded block** in dynamic space and a reference to it is stored into the original location. The encoding has associated costs for interrupting the operator and linearizing the structure, and the processor stalls until the entire write completes.

Similarly, when a processor reads a reference to a coded block from a cell in dynamic space, the operator at the processing site immediately retrieves and decodes the block, yielding a new, independent structure in local space that has the same form and internal relationships as the original. Again there are associated operator costs for the decoding, and the processor stalls for the duration. Repeated reads of the same coded block provide new, non-eq copies of the same structure.

This permits a QL program to conveniently mix globally mutable cells with independently mutable, structured values. Note that an application can choose to ignore this feature if it is required that only "pure" shared memory operations be used.

QL uses this facility when copying closures between processing sites. This is covered in section 2.1.

Named Structures

Named structures[5] are a useful data abstraction in Lisp. We provide a named structure type, `:named-shared-array`, that may be used with the Common Lisp `defstruct` macro to define data structures that reside in dynamic space. Named shared arrays are used to represent futures in QL, as described in section 2.1.

Support for Futures

Futures in QL are built from the futures provided by LAMINA. The latter are only created at processing sites and are managed by the operators at these sites. The primitive function `shared-read-if-present`, when called with (a reference to) a LAMINA future, will return only when the future has a value. The future may reside on any processing site and may be accessed from any processor. The calling process is always descheduled.

Another useful shared memory primitive is `shared-replace-conditional`. The form

`(shared-replace-conditional cell old new)`

atomically compares the contents of *cell* with *old*, and, if they are `eq`, replaces the contents with *new*. It returns `t` if the replacement took place; otherwise, it returns `nil`. The arguments *old* and *new* cannot be in local space. This primitive is used in QL to ensure that a future is determined only once.

2 QL

In this section, we describe the syntax and implementation of QL. All symbols denoting functions, macros and data structures are available in the `ql` package unless otherwise qualified.

2.1 Future, Delay, and Value

The fundamental means for evaluating an expression in parallel in QL is through the use of the `future` macro. The form

`(future expression)`

immediately creates and returns a future object that will hold the result of executing *expression* when it is available. Free variables in *expression* are lexically closed over and the closure is scheduled for execution according to the scheduling strategy in effect.

The closure environment formed by `future` is initially constructed in the local space of the calling process. It is then recursively copied into a coded block in dynamic space. When the closure finally executes, free variables that contained local structures in the caller's process will contain new,

isomorphic local structures in the called process. Therefore, any variables that are to be mutably shared between the caller and the callee must be represented as shared cells.

Futures are eagerly evaluated in the sense that a closure scheduled via `future` will always be run. QL provides a `delay` construct to serve the needs of demand-driven evaluation. The form

`(delay expression)`

otherwise behaves like `future`, but does not schedule the closure corresponding to *expression* until there is an attempt to determine the future.

Futures are not automatically determined in QL: there are no built-in strict functions (but see `qlset`). Instead, the function `value` is provided. The form

`(value potential-future)`

is the identity function on non-futures; otherwise, it blocks until *potential-future* has obtained a value—descheduling the calling process if it has not—and returns the result of recursively calling `value` on this value. In the case of a delay, the closure that will determine the value of the future is first scheduled if necessary. `value` always returns a non-future as its result.

A future may be shared between processes; it is guaranteed to be determined at most once. Any object may be tested to see whether it is a future (or delay) by the predicate `future-p`.

Implementation Details

A future in QL is represented by a named structure of type `:named-shared-array` with the following slots:

- `satisfied` : This is a flag that indicates whether the `future` has a value or not.
- `cell` : This is a LAMINA future which will contain a reference to the value of the future once it is available.
- `value` : The actual value of the future is stored here. Note that this value may itself be a future.
- `id` : This is an integer identifying the future for instrumentation purposes.

2.2 Qlet

A **delay** is represented by a named shared structure that includes the **future** structure. It has the following additional slots:

- **closure** : This contains a reference to the closure that will produce the value for the **future**. The reference itself always contains a coded block reference, which contains the closure.
- **requested-p** : This is a flag that is used to ensure that **closure** is scheduled at most once.

2.2 Qlet

Like QLAMBDA, QL provides the **qlet** macro for parallel lambda binding. The form

```
(qlet predicate bindings . body)
```

behaves like **let** if *predicate* evaluates to **nil**. If *predicate* evaluates to a non-**nil** value other than the symbol **:eager**, then each of the bindings is made in a separate process, and the calling process blocks until all the bindings have been made before continuing to evaluate *body*. Finally, if *predicate* evaluates to **:eager**, the bindings are started in separate processes but the calling process proceeds to evaluate *body*, blocking only when it requires the value of one of the bindings.

Note that the values of free variables that are closed over by each binding form in *bindings* are independently copied from local space into dynamic space if necessary.

qlet is implemented in terms of **future** and **value**.

2.3 Other Constructs

QL provides a number of other constructs that are based on **future** and **value**. Some of these are listed below.

- (**qspawn** *predicate . expressions*) behaves like **progn** if *predicate* evaluates to **nil**; otherwise it evaluates each sub-expression in *expressions* as a future. It always immediately returns **nil**.
- (**qmapcar** *function list . other-args*) calls *function* with each element of *list* and with *other-args* in a separate future, returning a list of the values of each future. The input list must be in local space, and the result is also constructed in local space.

- (*qmapc function list . other-args*) behaves like *qmapcar* but does not wait for the futures to be determined. It returns an undefined value. The input list must be in local space.

2.4 The Qlambda Process Closure

QL provides a *qlambda* construct for creating *process closures*[3]. However, unlike QLAMBDA's construct of the same name, there is no predicate to control whether the closure is run as a separate process³, nor can the closure be directly applied via the usual Lisp mechanisms.

The form

```
(qlambda lambda-list . body)
```

immediately creates a new process closure on a random processing site that, when later applied, will run the lexical closure defined by *body* with arguments bound according to *lambda-list*. Note that the values of free variables in *body* will be copied into dynamic space (as a coded block) if they are in local space.

qlambda blocks the calling process until the process closure has been created and returns a reference to it. It does *not* invoke the closure. Typically the result of calling *qlambda* is stored as a local or global variable for later use.

A process closure created by *qlambda* may be applied from any process by passing to it a list of evaluated arguments. The closure executes such calls by doing the appropriate lambda-binding, evaluating *body* in this context, and passing the result to the caller. Each application of the process closure is atomic with respect to the others. QL provides the following functions for closure application:

- (*apply-qlambda closure-reference args*) applies the process closure identified by *closure-reference* to the list of evaluated arguments in *args*, blocks until a value is returned from the application, and returns this value.
- (*funcall-qlambda closure-reference . args*) behaves analogously to *apply-qlambda*.
- (*invoke-qlambda closure-reference . args*) is a call for effect on the process closure. It passes a list of the evaluated *args* to the process closure identified by *closure-reference* and immediately returns without blocking.

³This is because a closure that is called by multiple processes needs to be atomic with respect to each call to ensure consistency. An ordinary closure does not have this property. Rather than associating a busy/free flag with a closure, we decided to restrict the semantics of *qlambda* to always create a new process.

2.5 The Task Scheduler

Implementation Details

Process closures are represented using *Flavors*[5] instances in (the simulation of) QL. Instances of the `qlambda` flavor have the following instance variables:

- **Arg-Queue** : This is a LAMINA shared queue into which references to the parameters and return address of each application of the `qlambda` are put (via `cu:shared-enqueue`). This queue also serves as the "handle" to the `qlambda`.
- **Closure** : This is the closure that is applied to the dequeued arguments whenever the `qlambda` is applied.

When a `qlambda` closure is applied, a new LAMINA future is created by the calling process to hold the result of the application. The future and the evaluated arguments are formed into a list which is moved into a cell in dynamic space at the associated memory site of the local processing site. The cell is queued on the argument queue of the closure and then the calling process blocks. Some time later the closure is applied to the (dereferenced) arguments and a new cell containing the returned value is stored as the value of the future. The calling process unblocks and reads this cell to obtain the result of the `qlambda` application.

2.5 The Task Scheduler

The closures requested to be scheduled via `future` and `delay` are called tasks. QL provides a scheduling strategy that runs these tasks in processes.

QL's current scheduler may be characterized as centralized and demand-driven. References to requested tasks are queued on a central shared queue. Free processors request tasks from this queue and run them locally. Processors are free if they have no runnable processes; processes that are blocked on futures are not runnable. There is no preemptive scheduling at a processor: a process runs until it either blocks or completes.

The function `initialize-ql` initializes the QL scheduler. It must be called by the application code during startup. The function `available-processors` is used at that time to set the pool of processors on which QL tasks may be run; by default, all processors are part of the pool. The global constant `***task-queue***` points to the `cu:shared-queue` of requested QL tasks. This queue is created on the processor at which the call to `initialize-ql` is made.

3 Instrumentation

QL is provided with instrumentation to study the performance of applications written in the language. In particular, an *instrument* of flavor *ql-instrument* may be used to visualize the performance of the system. Figure 3 shows the *panels* of such an instrument, described below.

Processor Queue History is a scrolling line panel displaying the available parallelism in the system over time. The horizontal axis depicts simulation time and the vertical axis shows the total number of runnable and running processes in the multiprocessor model.

Processor Queue Load is a scrolling bar panel displaying the **queue load** per processor over time. Queue load is defined as $1 - 1/(1 + N)$, where N is the number of runnable processes at the processor. The horizontal axis represents simulated time. The vertical axis represents the processors in the system. Gray shades depict the total process queue load per processor according to the mapping along the right margin of the panel. This panel is useful in seeing whether the system load is balanced.

Network Latency is a scrolling line panel displaying the communication latencies in the system. The horizontal axis shows the simulation time of packet arrivals at memory controllers and operators. The vertical axis shows the transmission delays experienced by these packets. Memory latencies are approximately twice the displayed latencies.

QL Task Activity is a scrolling text panel showing the activity of application tasks. Each line of text in this panel corresponds to a QL task and shows the following information:

- *ID* is the unique integer identifying the task.
- *Status* is the most recent status of the task: * indicates a running task, ? indicates a task that is blocked on a future, and ! indicates a terminated task.
- *Activations* is the number of times the task was scheduled to run.
- *Run* is the cumulative run time of the task in milliseconds. This does not include time accessing shared memory.
- *Stall/MemOps* is the cumulative time, in milliseconds, spent stalling on shared memory operations, and the total number of such operations performed by the task.
- *Blocked* is the total time, in milliseconds, for which the task has been blocked on futures. This does not include time spent waiting for processor resources.
- *Runnable* is the total time, in milliseconds, spent by the task in a runnable state, *i.e.*, waiting for processor resources.
- *Site* is the processing site at which the task is located.
- *Task* is the top-level closure of the task.

The lines are sorted according to the time the task was most recently scheduled.

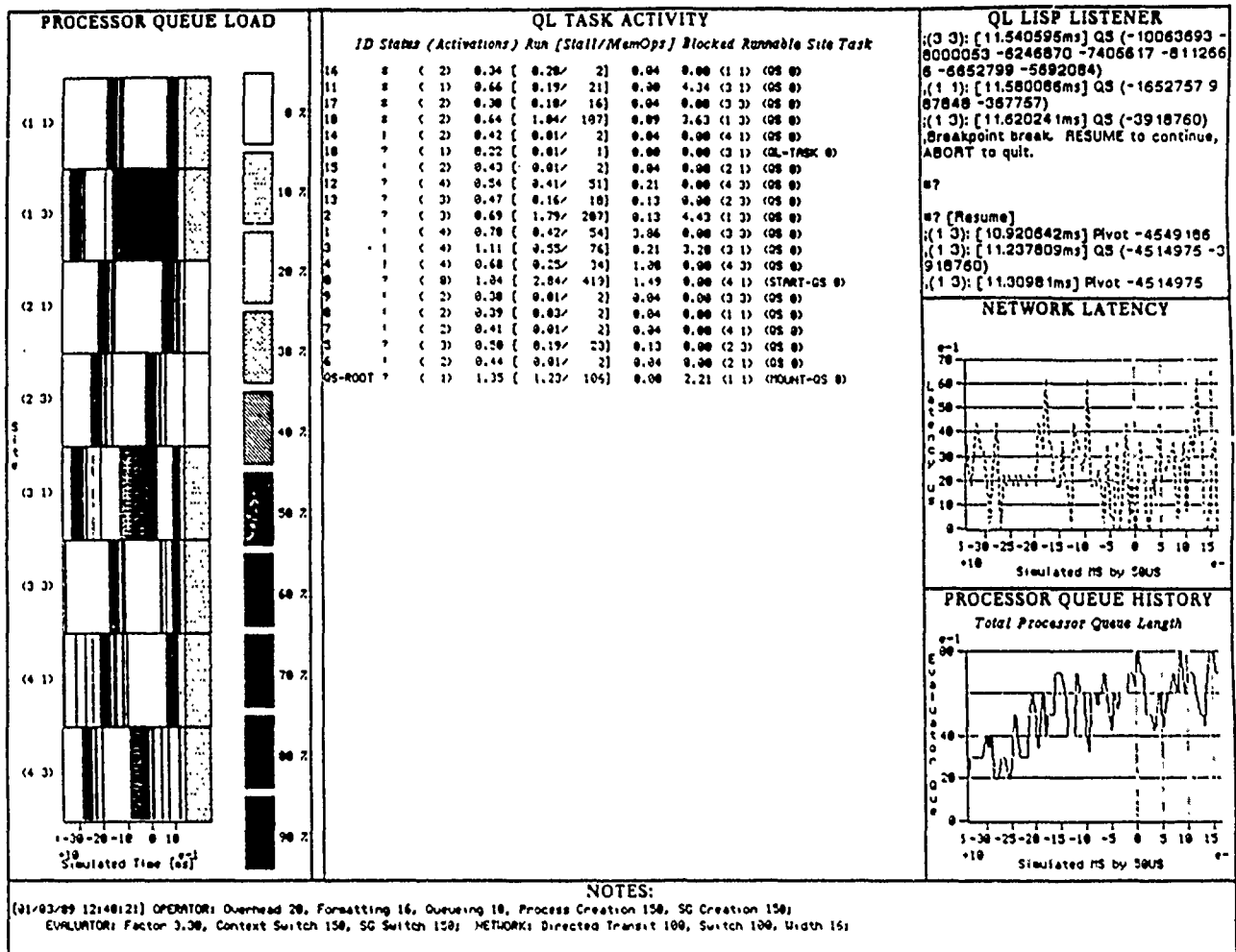


Figure 3: A QL Instrument

4 Current Status

QL is an operational shared memory Lisp simulation, although it is by no means a complete Lisp implementation. The non-uniform treatment of shared data structures (which are dealt with "by reference") and local data structures (which are dealt with "by value") somewhat modifies the semantics of traditional Lisp operations. This is particularly evident in the case of free variables that are closed over with `future` and `qlambda`—variables in local space are always copied and cannot be mutably shared. Furthermore, simulation efficiency concerns dictate that the Zetalisp functions that underlie QL are generally only applicable to local data: shared data structures must be manipulated via the LAMINA shared variable operations. The same concerns also lead to the values and property lists of symbols not being modeled as shared cells in dynamic space.

The most useful style of programming with QL is one that uses shared cells to represent *all* and *only* potentially mutable shared data. QL has been successfully used to construct a "blackboard" shell system for the development of concurrent expert system applications.

References

- [1] Bruce A. Delagi, Nakul Saraiya, Sayuri Nishimura, and Greg Byrd. An instrumented architectural simulation system. Technical Report STAN-CS-87-1148 or KSL-S6-36, Knowledge Systems Laboratory, Computer Science Department, Stanford University. January 1987.
- [2] Bruce A. Delagi, Nakul P. Saraiya, and Gregory T. Byrd. LAMINA: CARE applications interface. Technical Report KSL-S6-67, Knowledge Systems Laboratory, Computer Science Department, Stanford University, November 1987.
- [3] Richard P. Gabriel and John McCarthy. Queue-based multi-processing lisp. In *1984 ACM Symposium on Lisp and Functional Programming*, pages 25-44. Austin, Texas, Aug 1984.
- [4] Robert H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501-538, October 1985.
- [5] Daniel Weinreb and David Moon. *Lisp Machine Manual*. Symbolics, Cambridge, MA, 1981.

Design and Performance Evaluation of a Parallel Report Integration System

Nekul P. Saraiya

Bruce A. Delagi

Sayuri Nishimura

KNOWLEDGE SYSTEMS LABORATORY
Department of Computer Science
Stanford University
Stanford, CA 94305

and

DIGITAL EQUIPMENT CORPORATION
Palo Alto, CA 94301

This work was supported by DARPA Contract F30602-S5-C-0012, by NASA Ames Contract NCC 2-220-S1, by Boeing Contract W266875, and by Digital Equipment Corporation.

Design and Performance Evaluation of a Parallel Report Integration System ^{*†}

Nakul P. Saraiya and Sayuri Nishimura
Knowledge Systems Laboratory
Stanford University, Stanford CA 94305

Bruce A. Delagi
Digital Equipment Corporation
Palo Alto, CA 94301

Abstract

We describe the design and performance of a soft real-time report integration system written in a concurrent object-based programming language. We introduce the model that forms the foundation of the language—that of pipelines of asynchronously communicating objects executing data-driven, run-to-completion tasks—and discuss the mapping from the problem domain to the model through the design of the application. Using an instrumented architectural simulation system, we quantify the performance of the system over a range of one to 256 processors in terms of its sustainable input data rate. We show that the system approaches the limits imposed by task granularity and message handling costs when there is no contention for processing resources, and we study the effects of load imbalance.

1 Introduction

At the Advanced Architectures Project (AAP), we have been interested in speeding up the class of applications known as soft real-time *report integration systems* [10] through parallelism. This paper documents one of a number of efforts within the AAP to achieve this goal [2,15,18,19].

Our application is a system for interpreting preprocessed, passively-acquired radar emissions from aircraft. The system, called ELINT, is one component of a multi-sensor report integration system [26]. It was originally implemented as a "blackboard" [17] system using the AGE [16] shell, and later reimplemented in an experimental concurrent object framework called CAOS [21]. The results of that work [2] led to the version reported here.

^{*}This work was supported by DARPA Contract F30602-85-C-0012, by NASA Ames Contract NCC 2-220-S1, by Boeing Contract W266875, and by Digital Equipment Corporation.

[†]A condensed version of this paper was presented to the 1988 SIGPLAN Workshop on Object-Based Concurrent Programming [6].

Our programming language is a concurrent, object-oriented extension of LISP called LAMINA [7]. It promotes a programming model in which an application is decomposed into pipelines of asynchronously communicating objects that execute data-driven, run-to-completion tasks in response to messages. LAMINA is targeted to an ensemble machine [22] consisting of several hundred independent processor-memory pairs communicating via a low-latency network. Our work was done in the context of a simulation of such a machine [8], developed using an instrumented architectural simulator [9].

The rest of the paper is organized as follows. The next section introduces and characterizes the ELINT problem. Section 3 describes the LAMINA object-oriented programming model. Section 4 elaborates on the machine model that we used for our work, focussing on its support for LAMINA. Section 5 considers the mapping from the problem domain to the model by discussing the design of ELINT. We use the discussion to illustrate various techniques for organizing LAMINA applications to enhance performance. Section 6 describes our experiments to quantify the performance of ELINT. We show that the system approaches the limits imposed by task granularity and message handling costs in the absence of contention for processing resources, and we study the effects of load imbalance. Finally, section 7 offers some conclusions.

2 The ELINT Problem

ELINT correlates the radar emissions that are passively observed by multiple, mobile detection sites into the individual radar emitters producing those emissions. It tracks the emitters and then groups them into clusters that are tracking together. Finally, it hypothesizes the types and number of platforms (aircraft) in the clusters and infers their activity.

The inputs to ELINT are time-ordered streams of preprocessed *observations* from the detection sites. Each observation contains an indication of the current data timeslice, an identifier for the detected radar emitter, and information regarding its current signal characteristics such as quality, operating mode and line of bearing. The outputs of ELINT are real-time reports on the tracks, constituent platforms, and activities of the clusters in the monitored airspace. These reports may be used, for example, to maintain a "situation board" describing the status of the airspace.

Characterization of the Problem

ELINT exhibits a number of characteristics that are of interest when considering ways of organizing it as a parallel system.

- The system processes continuous, errorful, real-time input data.

- The input data describes relatively independent and persistent activities. Aircraft are independent entities (radar emitters somewhat less so), and they tend to be detected for long intervals, relative to the data period.
- The overall problem has an irregular, dynamic and data-dependent structure. Aircraft come and go over time, and their numbers, types, movements and intentions vary.
- Problem solution proceeds largely by “abstracting” the input data. Detected emissions are correlated into radar emitters that are further tracked and consolidated into aircraft.

This suggests that any parallel decomposition scheme would do well to exploit the potentially large amount of data parallelism inherent in the problem as multiple, loosely-coupled activities are monitored. Pipelining is also indicated in the data-driven abstraction process. Lastly, the persistence of activities makes an object-oriented representation attractive because it localizes the data-driven state changes, generates concurrent processes (i.e., objects) only as needed, and allows the cost of establishing a concurrent process to be amortized over the lifetime of the activity it models.

In the next section, we discuss an object-oriented programming model that evolved to efficiently realize these ideas.

3 The LAMINA Object Model

The LAMINA object programming model is founded on the notion of asynchronously communicating objects. An object, as used here, is a collection of variables—its *state variables*—manipulated by (and only by) a set of procedures—the *methods* associated with that object. Objects may be defined within a compiled class inheritance network; the current implementation uses the inheritance facilities of *Flavors* [25] (see [7] for details).

Objects communicate via *streams* [7], that are a generalization of futures [12,14], which represent the promise for a value. Unlike futures, which represent single values, streams represent sequences of values. The information sent to a stream over time builds the sequence associated with it. A consumer accesses the sequence by removing individual items from the front of the stream.

Each object has associated with it a distinguished stream that serves as its *task stream*. The information placed on this stream specifies tasks for the object; each unit of information is called a *message*. A message names a method to execute and includes the parameter values for the execution. A task that sends a message neither waits for the requested task to be performed, nor for an acknowledgement of the receipt of the message. Communication between objects is thus completely asynchronous.

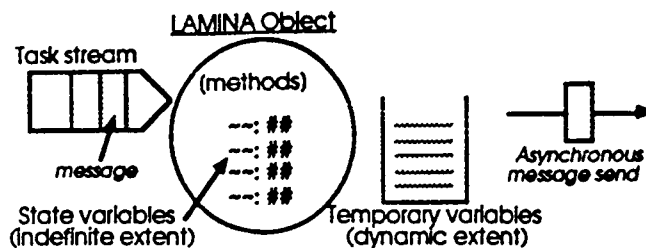


Figure 1: Message passing model

Most parameters included in a message are passed by copying; only streams and globally-shared static constants (*e.g.*, keyword symbols) are passed by reference. A sender may provide parameter values that are arbitrarily structured graphs composed of lists and arrays, and they are isomorphically reproduced at the receiving stream. This necessitates the *encoding* and *decoding* of messages at the sender and receiver, respectively, and the mechanism through which this is efficiently accomplished is discussed in section 4.

3.1 Computational Flow

As illustrated in figure 1, the messages arriving on an object's task stream specify tasks to be performed by that object. Every object has a *dispatch process* associated with it which removes and executes each message on its task stream in turn. Tasks usually mutate the state variables of the object and generate new messages. They have exclusive access to their environment (*i.e.*, state and temporary variables) during execution.

Tasks are *data driven* in that they are started only when all the needed information is available. Typically, a single message, in conjunction with the object's state variables, contains all the relevant information for a task. LAMINA also has mechanisms for scheduling tasks upon multiple message arrivals [7].

Tasks are generally intended to be accomplished as the stages of pipelines that organize the work performed by the objects of the application. In order not to block the pipeline, a task, once started, is *run to completion*.

3.2 Providing Atomicity

Although LAMINA provides the programmer with a run-to-completion model, there may be system reasons for preempting a task, for example, to handle a debug trap or because the task's run quantum has expired. When this occurs, the object does not execute any other tasks until the preemption is resolved. This prevents other tasks on that object from gaining access to the environment of the suspended task. However, since other objects may

execute tasks during this time, true atomicity can only be enforced if no state is shared between environments. The mechanism by which objects communicate ensures this.

LAMINA objects can never share state because they only communicate by exchanging messages containing *independent* copies of local structures. Furthermore, the state variables of an object are only visible to its own methods and are therefore only accessible within a private task. Thus the atomicity of operations on an object is preserved even in the presence of preemptions.

3.3 Scheduling

The order of messages on an object's task stream dictates the order in which its tasks are scheduled. Task streams may be specialized to override their default FIFO behavior, and, in addition, to impose either prioritization or sequencing on messages, and therefore task activations.

Prioritized, or ordered, task streams merge arriving messages with those already on the stream according to the numeric order keys included in the messages. Thus, the highest priority pending task is scheduled first. *Sequenced* task streams additionally restrict message removals (and, hence, task activations) according to their order keys, deferring out-of-order messages until after the tasks corresponding to the intervening messages in the sequence have been executed. In both cases, scheduling is programmer-specified and cooperative because the order keys must be supplied by the tasks originating the messages.

3.4 Continuations

A LAMINA message specifies a transfer of control to the named method in the destination object along with data relevant to continuing the computation at that point. It may thus be usefully viewed as specifying a *continuation* [23] for the task that originates the message.

A LAMINA continuation is most often determined and invoked *explicitly*. The continuation is determined either via computation within the originating task (using the contents of the state variables of the object) or by virtue of being passed in as a *client* parameter. In either case, its invocation is visible within the method code, as a message with the appropriate selector (and, perhaps, yet another client) that is explicitly sent to the chosen object.

When an explicit continuation is not convenient (*e.g.*, in performing a remote function call) an *implicit*, anonymous continuation can be used to capture the code and environment needed to later finish the local computation. LAMINA forms the continuation by copying any required bindings that are on the stack into a closure [24], which is deferred until further information (*e.g.*, the return value) is available.

An implicit continuation is not part of the original task's atomic execution; the task and its continuation are independently atomic. The original task is first completed and

its continuation is run some time later, when its requirements for additional information have been met. In the meantime other tasks are executed by the object, keeping the pipeline flowing. Since the continuation shares its spawning task's execution environment, its environment may be altered by other tasks on the same object while it awaits execution. The programmer must ensure that invariants are reestablished by the completion of each task and continuation.

3.5 Granularity and Storage Considerations

The internal state of a LAMINA object (contained in its state variables) is expected to occupy on the order of tens to hundreds of bytes, and task execution is expected to require on the order of hundreds to thousands of machine cycles. This perspective on granularity is important when considering means of supporting the LAMINA object model.

For example, if we assume that the support layer includes a virtual-memory system, then high-performance, physically-addressed caches imply a page size of a few kilobytes. If we further assume a page-protection-based stack-limit mechanism, then by maintaining a stack for each object we pay a factor of a hundred or so in the (physical) space overhead associated with task execution.

Since LAMINA tasks and their implicit continuations normally run to completion, their binding and control stacks are non-empty only during execution. Since task preemption is an exceptional condition, stack storage space is generally reuseable among all the tasks on a processor. This allows the underlying system to manage relatively fine-grained objects without foregoing efficient virtual memory and cache mechanisms.

3.6 Model Summary

In summary, LAMINA encourages programs to be decomposed into data-driven, run-to-completion tasks that are executed by objects organized as pipelines. Objects execute the tasks in response to messages arriving on their task streams, using their associated dispatch process. Only streams and universally-shared, static constants are passed by reference in messages; all other data, including structures, are passed by copying. Each task atomically manipulates the message contents and the state of the object before asynchronously sending new messages. Messages specify continuations, most often explicitly determined and invoked, and they usually activate tasks in objects further down the pipeline.

The LAMINA object model is similar to ACTORS [1] in that message arrival triggers computation and message arrival order is non-deterministic. However, it departs from ACTORS in a number of ways, primarily by trading off flexibility for efficiency.

- Not everything is an object. Predefined data types such as numbers, symbols, arrays

and cons cells exist as primitives, and operations on them do not entail message-passing. Although structures are passed by copying, they are locally mutable.

- Streams are first-class entities independent of objects. Objects may establish communications over streams other than their task streams. Streams may also be shared between objects as described in [7].
- Mutation is explicit. Unlike actors, LAMINA objects do not deal with state changes by specifying a *replacement* [1] actor for themselves, but rather explicitly manipulate their own state variables through assignment.

In the next section, we consider an architecture to support the LAMINA programming model.

4 The (Simulated) Message-Passing Machine

LAMINA is targeted to an ensemble machine consisting of hundreds to thousands of independent processing *sites* that are interconnected by a direct communications network. The network supports a low-latency, cut-through, point-to-point routing protocol that includes a multicast facility [3].

As shown in figure 2, each site consists of

- an *evaluator*, a general-purpose processor that is responsible for executing application code;
- an *operator*, a dedicated co-processor that is responsible for creating and accepting messages, and for scheduling evaluator processes;
- some *buffers*, which interface the operator to the communications facility;
- some *memory*, which is shared by the evaluator, the operator and the buffers at the site; and,
- some *network ports* and a dedicated *router*, which together form the communications hardware of the site.

These components are described in more detail below.

4.1 Communications Support

The processing sites are typically embedded in a low-dimensional network, such as a mesh or torus, with routing decisions independently made at each site.

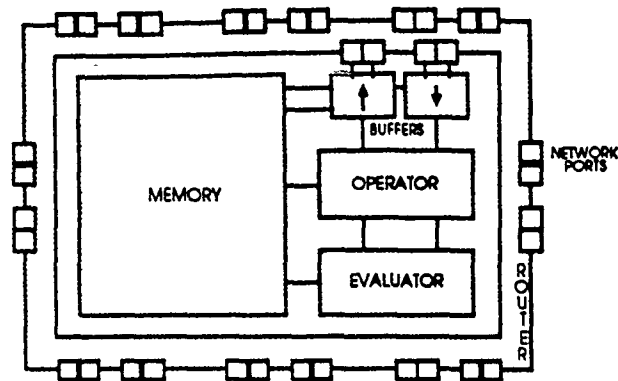


Figure 2: A processing site

The communications hardware at each site implements a dynamic, cut-through routing algorithm [3]. Cut-through routing means that a router makes a routing decision as soon as the first few bytes of a packet have been received [13], rather than after the entire packet has been buffered as in store-and-forward schemes. The decision is dynamic in our system because it is based on the availability of output ports at that time, thus adapting to prevailing network load conditions. In the exceptional case that no suitable port is available due to congestion, the router instead initiates a connection to the local buffer and the message is stored in its entirety in local memory for later retransmission by the operator.

With cut-through routing, the latency of a message is proportional to the sum of its length and the distance it covers in the network [5], as opposed to being proportional to the product of the two as with store-and-forward schemes. This results in a significant reduction in latency, making finer-grain LAMINA tasks cost-effective. It also makes locality less of an issue in object allocation.

Finally, the router provides direct support for multicast packets [3], allowing low-latency transmission of the same packet to multiple destinations while using common channels as far as possible.¹ This is a useful feature for LAMINA applications because it provides an efficient mechanism for messages that are multicast to replicated objects.

¹We have recently found that the multicast scheme that was used for our experiments can degrade unacceptably in the presence of even moderate load; in [4], we present an alternative approach with better load characteristics. Our experimental parameters for the work reported in this paper did not stress the multicast facility sufficiently for its effects to be significant.

4.2 Message-Handling Support

The operator provides the message handling facilities at a processing site. It is primarily responsible for accepting and generating messages in the form of packets. It also enables and schedules processes for execution in the evaluator.

As described earlier, LAMINA messages may contain values with arbitrary internal structure. These are to be passed by copying, and must be *encoded* before transmission. This function is performed by the operator. When the process running in the evaluator needs to transmit a message, it passes the operator a pointer to the data to be transmitted. The operator recursively traverses the data, linearizing it and relativizing internal pointers to produce a coded, relocatable copy. It forms this into a packet and transfers it to the network buffer for transmission to the remote site.

When the packet arrives at its destination, the operator at that site performs the reverse mapping, *decoding*, offsetting the relativized pointers to produce an isomorphic copy of the original data in local memory, within the address space of the receiving object.

The evaluator at the transmitting site is free to continue processing once it has passed the message data pointer to the operator. However, it must not mutate the data until the encoding is completed. Similarly, the evaluator at the destination site is not involved in message reception. The operator places the message on the targeted stream, and, if it is for a dormant LAMINA object, enables the appropriate dispatch process and passes it to the evaluator for execution.

4.3 Simulation Model and Parameters

We have used the SIMPLE architectural simulation system [9] to develop our machine model, called CARE [8]. Besides allowing one to flexibly specify models at any level of detail, SIMPLE provides extensive instrumentation facilities to visualize and study the performance of both the modelled architecture and the applications driving it. The feedback provided by the instrumentation was invaluable in refining the design of ELINT. SIMPLE currently runs on the Texas Instruments *Explorer* and Symbolics *3600* families of machines; we are engaged in porting it to Common Lisp [24] and the X window system [20].

CARE simulates the communications subsystem at approximately the register-transfer level. The operator is functionally simulated and the evaluator uses the base machine to actually execute and time the application code. Most components are parameterized, allowing a wide range of architectures to be modelled with little effort.

For the experiment reported here, the base system cycle was set at 100ns, so the evaluator was assumed to deliver upto 10 MIPS performance.² Process switching was assumed

²Depending on the expected ratio between peak and average pipeline occupancy, this might realistically represent approximately 7 MIPS delivered instruction execution rate.

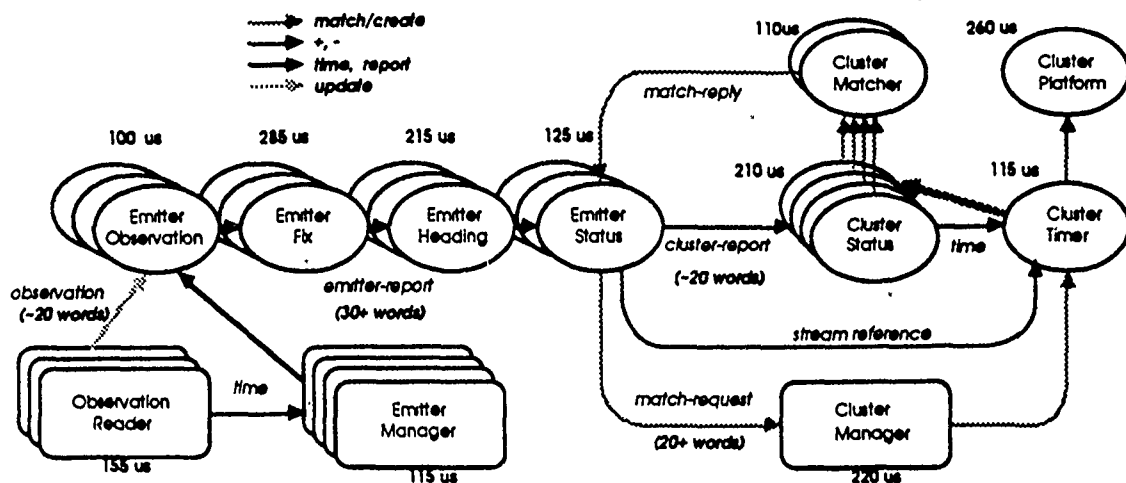


Figure 3: ELINT organization, with average task and message granularity

to take 150 cycles in each direction. Message creation and acceptance cost 100 cycles to interrupt the operator and setup internal registers and an additional 16 cycles per 32-bit word of data for encoding or decoding. Network channels were 1 bits wide, giving a channel bandwidth of 40 Mb/s. Finally, the system was configured as a torus with eight bidirectional connections at each site.

5 Application Design

ELINT's basic processing flow is data-driven. Every data timeslice, new observations are input into the system and either correlated with known emitters or used to create new emitters. The data contained in the observations are used to update the status and course history of the emitters. Any unclustered emitters with sufficient track data are matched against known clusters, and, failing that, formed into new clusters. Finally, data from the emitters are used to update the track, activity and platform history of their clusters.

5.1 Pipeline Organization

ELINT is naturally organized as parallel, data-driven object pipelines in LAMINA, as shown in figure 3.

Observation readers form the input interface of the system to the data collection sites. They read in time-tagged observation structures³, representing observed emissions, and pass

³Observations are stored in a data file, called a *scenario*, in our simulations.

these on to the **emitter observations** for the identified emitters. If the emission represents a new emitter they instead pass it to the **emitter manager** responsible, which creates a new one (if necessary) and so informs the readers.

Emitter observations buffer the observations until the end of the current data timeslice is detected. Thereafter, they compute confidence and status information about the represented emitter and form an **emitter-report** structure that they pass down the pipeline. Successive pipeline stages compute and attach track (*i.e.*, fix and heading) information to this report.

Emitter status objects link their respective emitters to an appropriate cluster, and, once clustered, propagate their reports to the cluster. Each group of **cluster status** objects implements a distributed database of the track and activity history of a cluster; this is used both to report on the cluster periodically and to match against the tracks of new emitters that attempt to cluster. Finally, whenever an emitter is added to or removed from a cluster, the associated **cluster platform** object adjusts its hypothesis on the platforms forming the cluster.

5.2 Replication

Replication is a useful means of relieving congestion. When feasible, it allows for wider pipelines and increased throughput. However, it is viable only to the extent that the replicated objects have no dependencies between them; otherwise, duplicated data must be kept consistent.

Early experience indicated that the **observation readers** were obvious candidates for replication, since the rest of the system was often data starved. **Emitter managers** are also replicated to scale with the size of the system; a simple modulo operation on an emitter's (integer) identifier is used to break dependencies while maintaining creation consistency.

Sometimes the benefits of replication outweigh the need to maintain consistency at all times, especially if the system has the capacity to detect and correct the inconsistency as part of its regular problem solving activity. This is the case with the **cluster status** objects which form the database of a cluster's track and activity history.

The database is partitioned by data time to ameliorate the bottleneck that results when multiple constituent emitters supply reports to a cluster every timeslice. Consequently, **cluster status** objects detect "emitter splits" (*i.e.*, emitters that were part of the cluster but whose tracks have now diverged from that of the cluster) in isolation. This can lead to inconsistent decisions as to the particular emitter that is split off, as a result of different message arrival orders at the individual objects (since the first arrival determines the inherited track of the cluster for that data time). However, at worst, too many emitters are split off and the system recovers because these emitters retry clustering with all extant clusters. The benefit of replication here is two-fold: besides reducing congestion at the cluster, the grain size of the match performed at each object during clustering is also reduced.

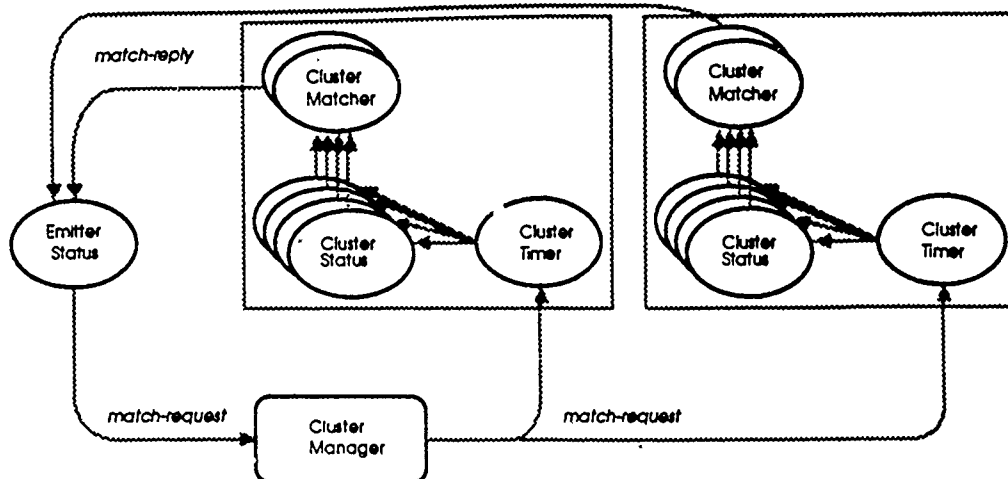


Figure 4: Clustering, showing a single emitter attempting to match two clusters

5.3 Clustering

An emitter that has been tracked for a sufficient period of time must either match an existing cluster or create a new one. This computation is organized as a distributed loop as shown in figure 4. During each iteration of the loop, the emitter attempts to match against those clusters that have been created since the last iteration, and the loop terminates when it successfully matches an existing cluster or causes a new one to be created. The process is described in more detail below.

An emitter status object that needs to cluster sends the cluster manager a *match-request* structure containing the recent track history of the emitter and a counter of the number of clusters that it has already (unsuccessfully) matched against. The latter maintains a private list of all extant clusters; it multicasts the request to the cluster timers of those clusters that have not been matched against, first incrementing the counter to reflect the total number of clusters. If, however, the counter shows that matches against all clusters have failed, it creates and initializes a new cluster for the emitter, for which the match always succeeds.

A cluster timer defers the request until it is coincident in time, a decision that is based upon the reports supplied by its constituent emitters. Thereupon it multicasts the request to its cluster status objects, specifying a selected cluster matcher as the intermediate client of the match to diffuse the fanin load. The cluster matcher collects the partial results of the match and then sends a summary of the results to the original emitter status client. The latter awaits either a successful reply from one cluster or unsuccessful replies from all the clusters that it attempted to match against during this iteration, and thus determines whether to terminate or continue the distributed loop.

The multi-level fanout and fanin trees allow for significant throughput in the match process. An emitter can concurrently match against multiple clusters, and each cluster can concurrently service multiple matches as well as perform each individual match in parallel. The major potential bottleneck in this scheme is the cluster manager; like Brown *et al.* in [2], we have not found it to be so in practice.

5.4 Time Consistency

Many of ELINT's inferences rely on time-ordered sequences of data. For example, one way the system computes an emitter's heading is by using its fixes from two consecutive timeslices. We saw earlier that LAMINA does not guarantee that message order is preserved. As a result, ELINT must maintain this order when necessary.

This is done by using sequenced or prioritized task streams in conjunction with timestamped messages. The "natural" timestamps are the data timeslice numbers for which the messages are relevant. An observation reader detects the token signifying the end of a timeslice in the input data and propagates it to all the emitter managers. These, in turn, pass the information on to their respective emitter observations, thus triggering the flow of computation through the system for that timeslice.

A sequenced stream can guarantee that messages will be processed in the prescribed order. However, its use is only viable when its producers are *a priori* known, because of the obvious difficulty in otherwise generating the order keys in sequence. Thus, for example, the sequenced approach works for the emitter heading objects because they only receive messages from their upstream emitter fixes.

In many cases, however, the dynamic communication patterns between objects, such as between emitters and clusters, make other techniques necessary. One approach is to approximate sequenced streams by using a single source to provide the sequence keys to a sequenced stream while allowing the other sources to view the stream as simply prioritized. This is what is done with the emitter observation objects: they receive observations from some subset of the many readers every timeslice but their sequencing information is provided by time messages from their respective managers.⁴ Other approaches are to use prioritized task streams or to explicitly defer out-of-order messages in "shadow" task queues maintained by the application code. Although none of these techniques can generally guarantee correct ordering as strictly-sequenced streams can, our experience is that they work well in ELINT, primarily because the producers of a stream are often approximately synchronized and they each usually generate messages in time order.

⁴Note that this scheme also generally preserves data consistency *within* a timeslice because, once the time message is received, all the relevant data are bundled up into a single timestamped emitter-report and started down the pipeline.

5.5 Object Allocation

ELINT uses only static allocation of objects to processing sites. Multiple allocators, the managers, independently make the allocation decisions at object creation time. The experiments reported here used no dynamic migration of objects to balance load.⁵

Random allocation performed well when the number of objects was far greater than the number of processing sites. A better scheme partitioned the processing sites among the object classes during initialization and allocated objects randomly within each block at runtime. The partition was based on the measured relative activity of the classes, using information provided by the underlying instrumentation system (see section 6.2). This strategy reserved resources for the expected critical path and reduced both the load variance over the processing grid as well as the interference between pipeline stages.

5.6 Summary and Discussion

The fundamental programming model of LAMINA appears well-suited to the ELINT domain. Object pipelines executing data-driven run-to-completion tasks match its basic real-time update cycle well. Replication and partitioning can be used in balancing pipelines, although they may introduce consistency problems. Often, the inconsistencies are only temporary and localized and the system can resolve them as part of its normal duties. Partitioning and pipelining can also be dynamically combined to form trees for highly concurrent many-to-many database matches.

Although inter-object data consistency problems generally do not arise because values cannot be shared in LAMINA, a run-to-completion model sometimes raises intra-object (*i.e.*, inter-task) consistency issues. This is particularly true when a chain of explicit continuations feeds back on itself,⁶ for example, in ELINT's distributed match loop. Our solutions typically entail explicitly keeping task-specific data (*e.g.*, flags) as required within the object and adding code to avoid or, failing that, resolve problems in each potentially interacting method. Thus, for example, if a represented emitter is found to have incorrect track data (due to observation errors) while it is in the process of fusion, the emitter status method that handles match replies from the clusters is responsible for invalidating the (incorrect) successful matches that may result.

Another area in which consistency is important is time. Sequenced task streams can be used to guarantee correct control of tasks that rely on time-ordered sequences of data, but they cannot be used in the presence of dynamic producer-consumer relationships. In these cases, prioritized task streams or explicit message deferrals can be effective. All these approaches work reasonably well in ELINT because its pipeline organization and its

⁵Dynamic migration is the subject of current research within the AAP [11].

⁶The implicit continuations described in section 3.4 are a degenerate case of this.

periodic update cycles keep the producers of a stream approximately synchronized.

ELINT was extensively refined as we gained insights into its behavior through the feedback provided by SIMPLE's instrumentation. This also helped in shaping its class-based allocation scheme.

In the next section, we discuss our experiments to quantify the utility of the concurrent system.

6 Performance Evaluation

We evaluated ELINT with respect to *correctness* and *timeliness*. We used correctness to assess the quality of ELINT's problem-solving and timeliness to determine its performance in terms of speedup. Our experiments and results are described below.

6.1 Correctness: Solution Quality

We evaluated correctness by driving ELINT with a number of scenarios that exercised all of its decision-making capabilities, including handling input data errors. In each case, we compared the outputs generated by the parallel ELINT implementation with those generated by a serial (and, by definition, correct) implementation. An analysis program used these outputs to assign grades in various categories as shown in table 1. The term "correct" used therein implies that the grading penalized inaccurate or missed reports of the relevant events.

Results

Once debugged, ELINT performed well with respect to correctness over a range of scenarios and data rates. The system always correctly identified important events such as emitter detection, tracking and fusion (clustering). However, at very high data rates, fusion latencies sometimes affected correctness. The system missed a few cluster reports because cluster creation was delayed until matches were known to have failed, and it also reported threats for as-yet unclustered emitters rather than via activity reports for their clusters. Note, however, that these lapses were always short-lived, since the fusion latencies were never more than a small number of timeslices.

The major reason for ELINT's good solution quality is its maintenance of creation and time consistency [2]. All creation requests are funnelled through "managers", which ensures that the objects in the application correctly reflect the real world that they are modelling. Time consistency is maintained as described in section 5.4, by controlling the execution order of time-dependent tasks via timestamped messages directed to sequenced and prioritized task streams.

<i>Measure</i>	<i>Comment</i>
<i>Recognized emitters</i>	Emitters correctly recognized.
<i>Observation ID errors</i>	Correctly reported observations for distinct emitters that were assigned the same ID by different detection sites.
<i>Observation inconsistencies</i>	Correctly reported observations for the same emitter that were assigned inconsistent emitter types by different detection sites.
<i>Emitter confidences</i>	Emitter confidences correctly reported.
<i>Emitter fixes</i>	Emitter fixes correctly reported.
<i>Emitter headings</i>	Emitter headings correctly reported.
<i>Emitter threats</i>	Emitter threats correctly reported.
<i>Clustered emitters</i>	Emitters correctly fused into some cluster.
<i>Co-clustered emitters</i>	Emitters correctly fused into the same cluster.
<i>Not co-clustered emitters</i>	Emitters correctly fused into distinct clusters.
<i>Timely clustering</i>	Clusters formed at correct time.
<i>Cluster platforms</i>	Correct platform hypotheses for clusters.
<i>Cluster activities</i>	Cluster activities correctly reported.

Table 1: ELINT correctness measures

6.2 Timeliness: Sustainable Data Rate

We used timeliness to measure the quantitative performance of ELINT. As with the solution quality analysis described in the previous section, we used the technique of analyzing the outputs of ELINT executions.

Every report that is output by ELINT (*e.g.*, an emitter fix) is pertinent to a particular data timeslice, since it is based on new observations that were (or were not) provided for that timeslice. Each timeslice represents one "tick" of an external clock (corresponding to the scan time of the detection sites) and thus also signifies the real time at which the data for that cycle was available to the system. Thus we defined the *latency* of a report to be the elapsed time from the start of the relevant timeslice to the time that the report was generated by the system.

Averaging the latencies for certain classes of reports (*e.g.*, emitter fixes) on a timeslice-by-timeslice basis gave us a means of testing whether the system could process the data at the rate at which it was provided—average latencies that increased over time meant that it could not. Thus, the *sustainable data rate* for a given scenario and processor population was the fastest rate (*i.e.*, smallest timeslice duration) at which data could be supplied to the system such that the latencies of these key inferences did not increase over time.

The inferences we chose to look at were those which were indicative of the "normal"

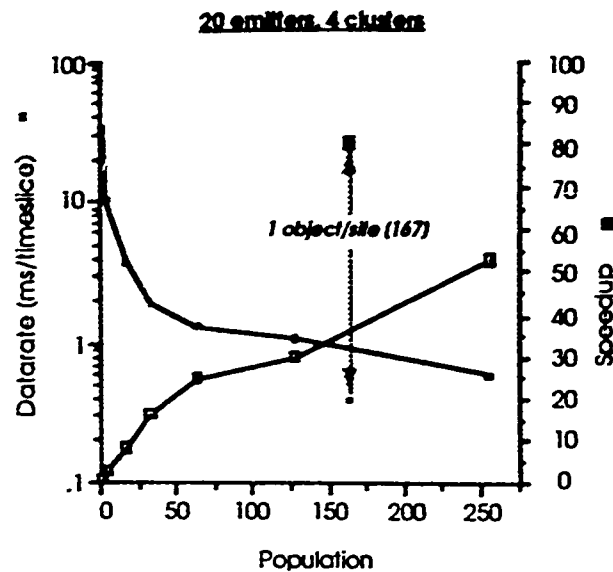


Figure 5: Speedup and Data Rate

processing duties of the system, namely, emitter and cluster track computations. These latencies were also a good choice from a pragmatic standpoint because they were sufficiently frequent for trends to be analysed programmatically without the problems induced by sparse data. We also used scenarios that contained a stable population of emitters and clusters so as to minimize the variations in observed latencies that could otherwise be caused by the sudden variations in system load.

Results

We measured *speedup* for a particular scenario by comparing the sustainable data rate for various processor populations with that for a single processor. The scenario reported upon here lasted forty timeslices and consisted of twenty emitters that formed four clusters. The results are shown in figure 5. Note that in all the cases, the good correctness levels reported in the previous section were attained.

With one object per site, ELINT could sustain a data rate of 400 microseconds per timeslice, a speedup of eighty over the serial case. The grain sizes of ELINT tasks and messages that are shown in figure 3 provide us with the explanation for this.

In particular, emitter observations have an average task granularity of 100 microseconds, and they each receive and process three messages per timeslice before initiating further pipeline activity. These are two update messages providing an observation from each of the detection sites in the scenario and a report message signifying the end of the current times-

lice. This creates a "hard" performance limit of 300–400 microseconds per data timeslice, which is approached in this case. Note that the limit is the maximum throughput of a single pipeline; it is independent of the width of the scenario, *i.e.*, the number of pipelines. Conversely, speedup is proportional to the number of pipelines that can operate in parallel, which is twenty in this case. The eighty-fold overall gain is realized because each individual pipeline gives an additional factor of four.

As the load on the sites increases with smaller processor populations the contention for site resources degrades performance, even with balanced loads, because pipelines must now run slower. A poor load balance additionally reduces performance, because pipelines can only go as fast as their slowest stage—the sustainable data rate of the system is now limited by the throughput of the worst loaded site.

Load variance across the sites accounts for sub-unitary speedup⁷. Non-ideal load distributions occur as a consequence of any realistic allocation scheme. Independent allocators, necessary to alleviate bottlenecks, cannot easily share a consistent view of the global load because of the distributed nature of CARE. Thus, for example, in the 64 site case, 6 managers used the class-based scheme to allocate 80 dynamic emitter objects over a pool of 25 sites.⁸ The best distribution here would have been one where 3–4 objects resided on each site; in actuality, two sites hosted 5 objects each, which limited the throughput of the system to 1.3 milliseconds per data timeslice. However, as noted earlier, a purely random allocation scheme resulted in even more inter-site variance than this.

Secondary factors affecting performance are:

- Dynamic object creation latencies are fixed costs (on the order of milliseconds) that become more dominant with higher data rates. Work that is deferred until creation is accomplished costs relatively more in terms of latency. Performance suffers since the system spends more of its time managing backed-up queues and playing "catch-up".
- The relative sizes of the processing pools allocated to various classes are set without prior knowledge of the scenario. The partition attempts to cover a range of possible scenarios and is thus never optimal with respect to any particular one.
- Similarly, the degree of replication of various objects is not scenario-specific, but is rather precomputed based on expectations of "typical" situations and available processing resources. Replicas and their processing sites may be under- or over-utilized for any given scenario.

⁷Unitary speedup is linear speedup with a slope of 1.

⁸Our equivalence of "load" with number of objects at a site is rather idealized—all objects need not have the same processing requirements. However, our problem decomposition and the persistent objects typical of ELINT make the approximation useful here.

- Finally, ELINT's pipeline stages are only approximately balanced. As shown in figure 3, task granularities range from 100–300 microseconds (with varying frequencies) and messages are typically 20–30 words in length. Also, secondary message paths feed back into the pipes, for example, during fusion. Both these factors reduce overall efficiency.

7 Conclusions

We have described LAMINA, a concurrent object-oriented programming language, and the design and simulated performance of ELINT, a soft real-time report integration system written in this language. We offer the following conclusions from our experiments.

- LAMINA's programming model of objects organized as pipelines is well-suited to the soft real-time domain, because it naturally exploits the data and pipeline parallelism inherent in the problem.
- A value-passing model in which asynchronous messages trigger data-driven, run-to-completion tasks can form a viable alternative to a call-return discipline. We have quantified the performance that results from keeping pipelines flowing, and we have shown how to preserve program solution quality within this model by keeping data consistent and controlling order-critical tasks.
- We have demonstrated that it is possible to obtain significant speedups for soft real-time systems within the above model. Our implementation of ELINT in LAMINA achieved a speedup of 80 in the absence of contention for processing resources, thus approaching the limits imposed by task and message-handling granularities. Since speedup depends on the number of pipelines that can operate in parallel, twenty in our experiments, we can expect performance to scale with wider scenarios that contain more aircraft.
- Load balance is an important factor in the performance of concurrent applications. This is especially true in real-time systems, where the focus is on throughput—an overloaded site limits the throughput of the entire system. Our class-based object allocation scheme performed better than random allocation, but still limited ELINT to a speedup of 53 with 256 processing sites. We expect that dynamic load balancing will be valuable in overcoming this loss in efficiency.
- System instrumentation is a critical tool in the development of concurrent programs. We found the feedback provided by the instrumentation in our simulator essential in refining the design of ELINT to break bottlenecks, balance pipelines, and evaluate load balancing schemes.

- Finally, we have described an architecture for a high-performance implementation of streams, which supports the low-latency delivery of data values to where they are needed and the efficient scheduling of consumers of the values. We believe that a cut-through network using adaptive routing forms a cornerstone of this support.

Acknowledgements

We would like to thank Harold Brown, for providing valuable guidance throughout the course of this work, and Max Hailperin, for doing much to improve the correctness of our application and the repeatability of our simulations. We would also like to thank the members of the Advanced Architectures Project for providing a stimulating research environment, and the staff of the Symbolic Systems Resources Group of the Knowledge Systems Laboratory for providing excellent support of our computing environment. Special thanks to Ed Feigenbaum for his work in support of the Knowledge Systems Laboratory and the Advanced Architectures Project. Max Hailperin provided valuable comments that improved the presentation of this paper.

References

- [1] Gul Agha. An overview of actor languages. *SIGPLAN Notices*, 21(10):58-67, October 1986.
- [2] Harold D. Brown, Eric Schoen, and Bruce A. Delagi. An experiment in knowledge-based signal understanding using parallel architectures. Technical Report STAN-CS-86-1136, Department of Computer Science, Stanford University, October 1986.
- [3] Gregory T. Byrd, Russell Nakano, and Bruce A. Delagi. A dynamic, cut-through communications protocol with multicast. Technical Report STAN-CS-87-1128, Department of Computer Science, Stanford University, September 1987.
- [4] Gregory T. Byrd, Nakul P. Saraiya, and Bruce A. Delagi. Multicast communication in multiprocessor systems. Technical Report KSL-88-81, Knowledge Systems Laboratory, Stanford University, December 1988. To appear in *1989 International Conference on Parallel Processing*.
- [5] William J. Dally. Wire-efficient VLSI multiprocessor communications networks. In *Advanced Research in VLSI: Proceedings of the 1987 Stanford Conference*, pages 391-415. The MIT Press, 1987.
- [6] Bruce A. Delagi and Nakul P. Saraiya. ELINT in LAMINA: Application of a concurrent object language. *SIGPLAN Notices*, 24(4):194-196, April 1989.

- [7] Bruce A. Delagi, Nakul P. Saraiya, and Gregory T. Byrd. LAMINA: CARE applications interface. In *Third International Supercomputing Conference*. Information Sciences Institute, 1988.
- [8] Bruce A. Delagi, Nakul P. Saraiya, Gregory T. Byrd, and Sayuri Nishimura. CARE User's Manual. Technical Report KSL-88-53, Knowledge Systems Laboratory, Stanford University, June 1988.
- [9] Bruce A. Delagi, Nakul P. Saraiya, Sayuri Nishimura, and Gregory T. Byrd. An instrumented architectural simulation system. In *Artificial Intelligence and Simulation: The Diversity of Applications*. The Society for Computer Simulation International, 1988.
- [10] John R. Delaney. Multi-system report integration using blackboards. In Lee S. Baumann, editor, *Proceedings of Expert Systems Workshop*, pages 179-184. DARPA, Science Applications International Corporation, April 1986.
- [11] Max Hailperin. Load balancing for massively-parallel soft-real-time systems. In *Frontiers '88: The Second Symposium on the Frontiers of Massively-Parallel Computation*. IEEE Press, October 1988.
- [12] Robert H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4), October 1985.
- [13] P. Kermani and L. Kleinrock. Virtual cut-through: A new computer communication switching technique. *Computer Networks*, 3:267, 1979.
- [14] Henry Lieberman. Thinking about lots of things at once without getting confused: Parallelism in Act 1. AI Memo No. 626, MIT AI Laboratory, 1980.
- [15] Russell Nakano, Masafumi Minami, and John Delaney. Experiments with a knowledge-based system on a multiprocessor. In *3rd International Supercomputing Conference*. Information Sciences Institute, 1988.
- [16] H. Penny Nii. An introduction to knowledge engineering, the blackboard model, and AGE. Technical Report HPP-80-20, Heuristic Programming Project, Stanford University, March 1980.
- [17] H. Penny Nii. Blackboard systems. *AI Magazine*, 7(2-3), 1986.
- [18] H. Penny Nii, Nelleke Aiello, and James Rice. Experiments on Cage and Poligon: Measuring the performance of parallel blackboard systems. Technical Report KSL-88-66, Knowledge Systems Laboratory, Stanford University, February 1989.

- [19] Alan Noble and Chris Rogers. AIRTRAC Path Association: Development of a knowledge-based system for a multiprocessor. Technical Report KSL-88-44, Knowledge Systems Laboratory, Stanford University, 1988.
- [20] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79-109, April 1986.
- [21] Eric Schoen. The CAOS system. Technical Report STAN-CS-86-1125, Department of Computer Science, Stanford University, 1986.
- [22] Charles L. Seitz. Experiments with VLSI ensemble machines. Technical Report 5102, Department of Computer Science, California Institute of Technology, 1983.
- [23] Guy L. Steele Jr. LAMBDA: The ultimate declarative. AI Memo No. 379, MIT AI Laboratory, November 1976.
- [24] Guy L. Steele Jr. *Common Lisp: The Language*. Digital Press, 1984.
- [25] Daniel Weinreb and David Moon. Flavors: Message-passing in the Lisp Machine. AI Memo No. 602, MIT AI Laboratory, 1980.
- [26] Mark Williams, Harold Brown, and Terry Barnes. TRICERO design description. Technical Report ESL-NS539, ESL, Inc., May 1984.

SIMPLE/CARE
An Instrumented Simulator
for Multiprocessor Architectures

Nakul P. Saraiya

Bruce A. Delagi

Sayuri Nishimura

KNOWLEDGE SYSTEMS LABORATORY
Department of Computer Science
Stanford University
Stanford, CA 94305

and

DIGITAL EQUIPMENT CORPORATION
Palo Alto, CA 94301

This work was supported by DARPA Contract F30602-85-C-0012, by NASA Ames Contract NCC 2-220-S1, by Boeing Contract W266875, and by Digital Equipment Corporation.

SIMPLE/CARE

An Instrumented Simulator for Multiprocessor Architectures *

Nakul P. Saraiya and Sayuri Nishimura
Knowledge Systems Laboratory
Stanford University, Stanford CA 94305

Bruce A. Delagi
Digital Equipment Corporation
Palo Alto, CA 94301

Abstract

Simulation of multiprocessor systems at an architectural level can offer an effective way to study critical design choices if: (1) the performance of the simulator is adequate to examine designs executing significant code bodies, rather than just toy problems or small application fragments, (2) the details of the simulation include the critical details of the design, (3) the view of system operation presented by the simulator instrumentation leads to useful insights on the problems with the design and with the applications driving it, and (4) the simulation system is flexible enough to allow one to easily ask unplanned questions that require making changes either in the design or in its measurement. This article describes SIMPLE/CARE, an instrumented simulation system for studying multiprocessor architectures, that was designed to meet these goals.

1 Introduction and Overview

Simulation systems are often developed in the context of a particular problem. To a degree, this is true for SIMPLE, a general-purpose modelling system, and CARE, the multiprocessor architecture simulator that runs on SIMPLE.¹ The problem motivating the development of SIMPLE/CARE was the performance study of multiprocessor systems composed of many hundreds of elements executing a set of signal interpretation applications that were to be implemented using several alternative programming formalisms [9].

This problem offered a set of constraints that governed the design of SIMPLE/CARE.

*This work was supported by DARPA Contract F30602-85-C-0012, by NASA Ames Contract NCC 2-220-S1, by Boeing Contract W266875, and by Digital Equipment Corporation.

¹SIMPLE is a descendent of the PALLADIO VLSI design system [3], that has been optimized for the subset of PALLADIO's capabilities relevant to hierarchical design capture and simulation. SIMPLE was originally developed in Zetalisp [14]; it currently uses Common Lisp [13] with Flavors [15]. A version of SIMPLE/CARE that runs on Texas Instruments *Explorer* workstations is currently available; a forthcoming release will use the X window system [12]. For more information, contact saraiya@sumex-aim.stanford.edu.

- The kinds of multiprocessor system components that would be needed and the ways in which these would be composed into complete systems was initially difficult to bound. This meant that component models would be modified and elaborated over time as design concepts evolved.
- It was evident that instrumentation requirements were similarly fluid. Results from early simulation runs were likely to identify alternative aspects of system operation that *should* have been monitored, but were not. Further, since the simulator was to be used by system architects and applications programmers alike, their individual needs for detail in the view of system operation had to be satisfied. It was thus important that instrumentation could be varied both rapidly and independently of the system models.
- The applications represented significant bodies of code, so simulation run times had to be minimized. This meant that some simplifications in system models were indicated. On the other hand, the interactions of multiprocessor system elements were the least understood aspect of system operation. Thus, it was desirable that the system models capture the details of these interactions; otherwise, simulation results would be suspect.

These constraints yielded the primary requirements for SIMPLE and CARE. For the former, it was that SIMPLE should offer significant flexibility with regard to the specification of system models and their instrumentation, while maintaining efficiency in simulating these models. For the latter, it was that, in order to accomplish runs with acceptable elapsed times, CARE should particularly focus on the details of a multiprocessor system's communications and scheduling support facilities: aspects of system execution critical to multiprocessor (as opposed to uniprocessor) operation.

The remainder of this section outlines how the organizations of SIMPLE and CARE contribute towards meeting these goals.

1.1 The Organization of SIMPLE

SIMPLE provides flexibility in specifying system models by partitioning issues of *system functionality* and *system instrumentation* into separate, largely independent domains of consideration. In both areas, SIMPLE further partitions concerns as shown in figure 1 and described below.

System Function

The principal abstraction supported by SIMPLE to specify system models is the *component*. A component represents a fragment of system functionality by encapsulating some private

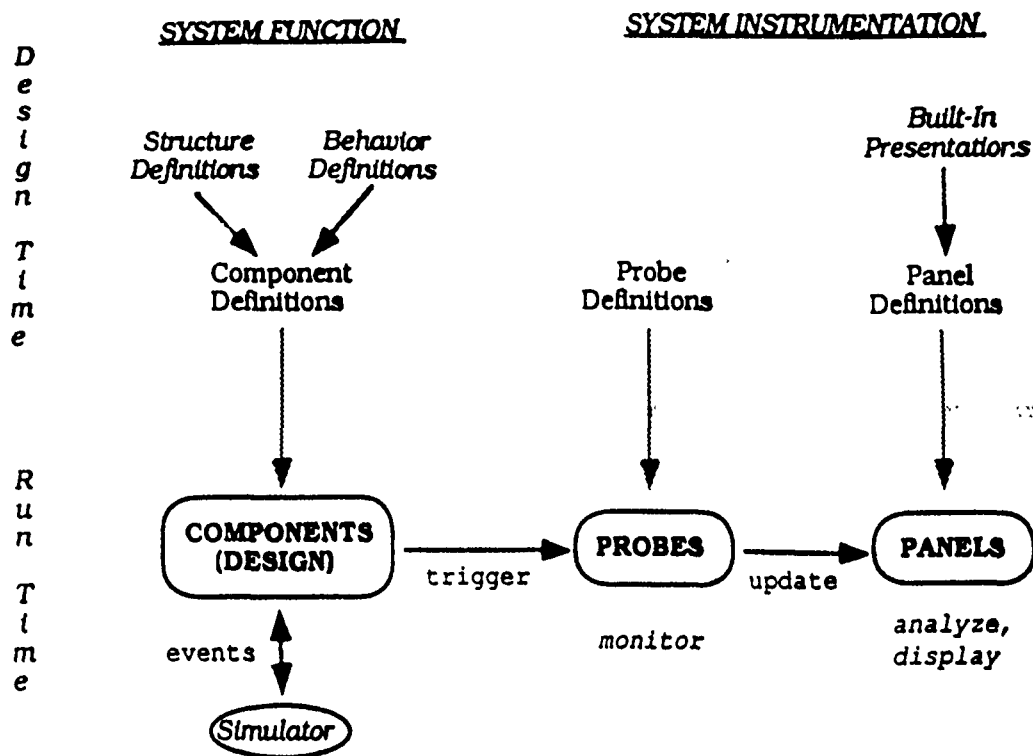


Figure 1: Simulator Organization

state along with the procedures that define how that state changes over time. A component is therefore naturally represented by an object.² The component abstraction partitions the design along well defined boundaries since, by and large, components interact only through their defined *ports*. Connections between components terminate at such ports so that, during a simulation run, assertions about the state of a port of one component can be directly translated to assertions about the state of a connected port of some other component.

System structure defines how components are combined to form a larger system. This is specified incrementally, as definitions for each component type which describe the sub-components (if any) a component of that type contains and how their ports are to be interconnected. Optional definitions for geometric layout and routing allow the designer to

²In fact, much of SIMPLE/CARE's flexibility and power derives from its implementation base of Flavors, which provides a mature, powerful object-oriented programming system with extensive capabilities for multiple inheritance and method combination. Flavors permits the evolutionary development of software 'libraries' for all aspects of the system design without sacrificing performance.

view the structure graphically. These specifications are captured as procedures, allowing efficient, parameterized and programmable structure generation [2]. A complete *design* is 'constructed' before a simulation run by the recursive generation of its parts, yielding a hierarchical network of interconnected components.

Component behavior defines how the state of a component changes over time. Behavior definitions are encapsulated as procedures relevant to each type of component, and can thus be developed mostly in isolation, provided interfaces are maintained. Behavior code is responsible for handling *events*—time-tagged state changes to a component's ports and internal state variables during a simulation run—in order to generate the local state changes 'caused' as a consequence.

System Instrumentation

Every component automatically includes support for instrumentation because every component inherits the basic functionality required for monitoring it and for maintaining its organizational relationships with the instrumentation system. This allows instrumentation to be introduced into the design *non-intrusively*, that is, without changing model function, and *incrementally*, that is, as interesting aspects of a component's operation are identified.

SIMPLE factors system instrumentation into the details of *data capture*, *data analysis*, and *presentation*. This allows for the flexible intermixing of different capabilities for each of these concerns.

Component probe definitions specify what data should be captured for each component type. There may be several probe types for a component type, each appropriate to measuring a different aspect of the component's operation. Probes may make use of pre-defined modules to accomplish certain types of calculations (for example, moving averages) on captured data.

Panels bring together the data analysis and presentation aspects of SIMPLE's instrumentation system. They specify how the data supplied by probes is to be transformed through analysis, and how the results are to be displayed. SIMPLE has a basic library of *presentations*: class definitions which represent particular display styles such as histograms, intensity maps and scrolling line plots. It also provides a library of procedures to accomplish standard data analysis operations.

A panel is defined by customizing the appropriate presentation class with descriptions affecting its graphical appearance, such as legends and color maps, along with *interface specifications*: expressions using an augmented Lisp syntax to describe probe types, data transforms, and displayed quantities. Defined panels may then be aggregated into an *instrument*, which associates a named type and a screen layout policy with the collection of panels.

Instrumentation is 'attached' to a design before a simulation run by simply instantiating

the appropriate instrument type with the design as a parameter. The instrument's panels are created and their corresponding interface specifications are then compiled into data structures and code that will accomplish the panel analysis and transformation operations. The required probes are also attached to components at that time. The end result is an *instrumented design* that ties together instances of components, probes and panels for the simulation run.

1.2 The Organization of CARE

At the base level, CARE provides a library of multiprocessor components such as network interfaces, busses, processors, message coprocessors and memory controllers. These can be composed into a number of standard system configurations, such as toroidal networks or systems of hierarchical busses. Most components are parameterized, allowing variation in performance characteristics such as cycle times and channel widths, as well as choices on other aspects of system behavior, such as routing algorithms.

To satisfy the need for detail required in modelling multiprocessor system element interactions, the definition of network components is fine enough to capture each of the many operations that accomplish cut-through message routing [8] of a packet of data in a torus network. To satisfy the runtime requirements of simulating complete applications, the processor models are coarse enough (and thereby fast enough) to ignore the details of simple processor operations that affect system operation only through their timing. Instead, this timing information is captured during the simulated execution of concurrent programs by dynamically running purely sequential segments of application code on the underlying machine and measuring their execution time.

Concurrent Programming Models

CARE defines parallel programming language extensions, collectively called LAMINA, for object oriented, shared variable and functional programming models [6]. The primitive mechanisms that support these language models are encapsulated within component definitions, but are decoupled from the underlying information flow control governing component behavior.

Component flow control actions deal generically with gating information between local ports and state variables, that is, with communicating information, independent of its content. Language support actions, on the other hand, create and manipulate information based solely upon its content; they play no part in communicating it between components. This separation of functionality allows the study of alternative communication protocols or topologies without modification to language interfaces and applications. Further, new

language interfaces may be defined or existing ones changed without redefining the communications protocol used by the system components.

Instrumentation

CARE supplies a library of probe, panel and instrument definitions corresponding to its particular multiprocessor systems and programming models. For example, one CARE system architecture is a message-passing multicomputer that executes application programs using the object-oriented LAMINA extensions [6]. An instrument for this system (see figure 14) has probes which monitor the critical operations performed on messages both by application data objects and by the resources of the underlying multiprocessor. These drive panels that display loads and latencies at the architectural as well as application levels.

1.3 Using SIMPLE/CARE

The primary users of SIMPLE/CARE are system architects and application programmers. The system architect develops multiprocessor component models and instrumentation through a set of design time interactions with the simulation system. The application developer, in turn, writes parallel code using the LAMINA language extensions or higher level frameworks derived from these [1, 10, 11]. All the definitions—for models, instrumentation, languages, and applications—are compiled and loaded into the Lisp environment. Incremental compilation, supplied by the environment, allows changes in these definitions have immediate effect, even during a simulation run, which is an important capability during debugging.

The user starts a simulation by first instantiating a design corresponding to the particular architectural model under study. The user then chooses a particular instrument and attaches it to the generated design, so that the instrument panels appear on the workstation screen. Another call then 'loads' the application program into the simulated multiprocessor and gets it running, at which time the instrument panels begin to dynamically display the chosen system performance measures. The user is free to interrupt the run both via the keyboard or via breakpoints inserted into the application or model codes. Menu-driven interactions allow variation of component model parameters as well as control of the instrumentation.

2 Building System Models

A system model or *design* is defined in SIMPLE by specifying its intended structure and behavior. As described earlier, this specification is organized around the components that form the system. In this section, we discuss the means by which system models are formulated in terms of components.

2.1 Structure

A system structure consists of a hierarchically-organized collection of typed components, as shown in figure 2. Defining such a system structure in SIMPLE involves defining each type of component and describing its contribution to overall system structure in terms of its subcomponents and their relationships.

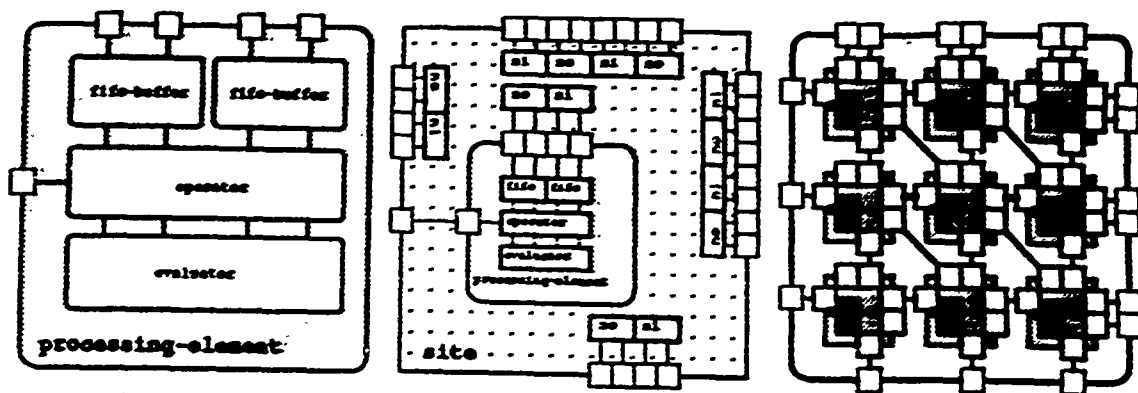


Figure 2: Hierarchical Composition

Defining Component Types

A component's type determines its private structure, that is, the set of attributes that make up the component. SIMPLE provides the `defcomponent` macro to define a type (or class) of component.³ This includes specification of its inheritance, and of the named slots, or *instance variables*, present in an instance of the class. Although slots may be used for any purpose, they primarily represent the *state variables* that are required for generating history-sensitive component behavior.

The code in figure 3 shows a fragment of the definition of a CARE operator component type. In this example, an operator is made to inherit from `debug-history-mixin`, thus gaining functionality for keeping a history of events during behavior debugging. The `Status` slot is a state variable that contains a symbolic representation of the run time state of the component. The `Pending-Operations` slot contains a complex data structure (a queue) that will be managed by the component during its operation.

³This macro has a straightforward translation to the underlying `defflavor` [15] construct which defines a *flavor*. It also generates definitions for procedures to initialize and reset the slots of a component of this flavor.


```

(defcomponent OPERATOR (debug-history-mixin)
  ((Status ; state variable
    :documentation "Current status: ready/busy/servicing"
    :initform 'ready :resetform 'ready)
   (Pending-Operations ; state variable
    :documentation "Data movement operations requested"
    :resetform (make-queue)) ; fifo
   ...)
  ...)

```

Figure 3: Definition of a Component Type

Defining System Structure

System structure in SIMPLE is built up through the incremental combination of components. Components can form a larger structure through composition and interconnection.

As shown in figure 2, at the base level are primitive components that have no defined structure beyond their ports. An operator is an example of such a component. Composite components, such as a *site*, additionally contain subcomponents as parts; parts may of course be primitive or composite. Composite components may also have function beyond what can be inferred strictly from their composition.

Composite components determine the interconnections between the ports of their individual subcomponents, and, further, the connections between their own ports and those of their subcomponents. Connections thus establish pathways for information to propagate between ports: both within and across hierarchical boundaries. Thus, the top-level composite forms the system structure, or design, under study.

SIMPLE originally captured component structure graphically and interactively, through the menu actions and mouse gestures supplied by a structural editor. Defined component subsystems would then be placed in a 'library' for later reuse. It turned out, however, that this approach was sometimes inconvenient. Furthermore, a database distinct from the underlying Lisp type database had to be maintained. Therefore, SIMPLE now represents structural information procedurally. A procedural representation permits efficient, flexible, and parameterized structure generation. It is particularly useful for automating the construction of the largely replicated system structures that characterize multiprocessor architectures.

A component's structure is specified as a method (that is, a procedure relevant to the type of the component) that is executed by SIMPLE's component instantiation protocol. The method uses built-in SIMPLE functions that create ports, subcomponents and connections to generate a component's structure. SIMPLE provides additional functions that allow the

description of the structural geometry of the component. In effect, then, these functions form the primitives that allow the construction (and querying) of a database of component objects. The component instantiation protocol accomplishes the creation of a system structure in a depth-first fashion. Components construct subcomponents, which in turn construct *their* subcomponents, and so on until primitive leaf components are created.

To illustrate this approach, consider the code in figure 4 that might define the structure of the processing-element shown in figure 2. In this example, *ev*, *op*, *ibuf*, and *obuf* are variables local to the structure definition.

```
(defstructure PROCESSING-ELEMENT (&aux ev op ibuf obuf)
  ;; Construct subcomponents and store into local variables
  (setf ev (part 'evaluator 'evaluator)
        op (part 'operator 'operator)
        ibuf (part 'buffer-in 'fifo-buffer :depth 10)
        obuf (part 'buffer-out 'fifo-buffer))
  ;; Construct ports
  (in 'network-packet-in) (out 'network-status-out) ; for ibuf
  ...
  ;; Establish connections between ports
  (conn (port? 'network-packet-in) (port? 'packet-in ibuf))
  (conn (port? 'network-status-out) (port? 'status-out ibuf))
  ...
)
```

Figure 4: Defining Component Structure

Subcomponents. Subcomponents are created via the *part* construct, which takes as arguments a name for the part, its type, and, optionally, parameters to customize the creation of the component. Thus, in figure 4, *ibuf* represents a *fifo-buffer* component named *buffer-in* which can hold no more than ten items. Subcomponents can be accessed by name through the *part?* function; here, however, they were stored into local variables for convenience. They may also be stored into predefined component slots or into data structures accessible via slots, so that they are easily accessible to behavior code.

The optional keyword parameters passed in to the *part* primitive are delivered to both the underlying Flavors instance initialization method as well as to the subcomponent's structure generation method. Each may then choose to ignore or use these arguments in customizing the creation of the new component. For example, parameters specifying dimensionality and connectivity might be among those passed to a subcomponent that generated a network of nodes organized into a grid topology.

Ports. Ports are classified by SIMPLE as either for input or output. Their corresponding constructors are `in` and `out`, both of which accept a name for the new port as a parameter. Thus, in figure 4, the call to `in` creates and returns an input port named `packet-in`. Ports can always be retrieved by name through the `port?` function. An optional argument to this function identifies the port's component; the calling component is the default.

Connections. Connections are established through the `conn` function, which takes two ports as arguments. Connections between subcomponents are unidirectional and must be between disparate types of ports: from input ports to output ports. Conversely, connected ports on a subcomponent and its superior must be of the same type, so that information may flow up and down the hierarchy. The information on a connection will be handled by the lowest component in the hierarchy that has an input port accessible via the connection.

Geometry. SIMPLE allows the structure generation code to be embellished with optional descriptions of the geometry of the component structure. Components can then be inspected via a graphical previewer, which facilitates debugging. To accomplish this, SIMPLE provides constructs to define the rectangles representing components, to place ports around the perimeter of the rectangle, to route connections in Manhattan space, and to place, group, align, and geometrically transform subcomponents.

2.2 Behavior

A component is essentially a state machine with a notion of time. Its behavior defines the causal and temporal progression of its states and relates this with the rest of the system via its ports. System behavior is therefore no more than the composition of the behaviors of its components.

Events signify the temporal state changes in the simulated system, in terms of the changes in the values of the ports and state variables of the system's components. They make the simulation of large, complex systems tractable, by exploiting the property that only a small fraction of the system state actually changes at any instant in time. This makes it more efficient to keep track of these changes and to compute their consequences, than to recompute the state of the entire system at every time step. An *event-driven simulator*, such as SIMPLE's, maintains the temporal relationships between events so that time always moves forward.

Within this framework, the behavioral specification of a component is formulated in terms of its responses to the events relevant to it. These responses may include state changes caused in the simulated future, that is, consequent events to be handled by the simulator, as well as direct operations on component state. The assertion of consequent

events and the responses to them (involving further consequences) drives the simulation. When there are no more events to handle, the simulation is complete.

To maintain modularity in a simulated system, a component's responses to events should generally be local to it. Consequent events involving a component's output ports are translated by the simulator into events involving the connected input ports of other components. Hence, the effects of a local change propagate between components along the connection paths defined by the system structure. Sometimes, however, a direct, non-local operation on a related component (for example, a subcomponent) might be appropriate. SIMPLE does not prohibit the modeller from accomplishing this.

SIMPLE captures component behavior procedurally, as the definition of a method on the component class. This method is charged with *asserting* and *processing* the events that drive the simulation.

Asserting Events

In concrete terms, an event in SIMPLE is a record that represents a single state change to the simulated system. It stipulates the component affected, its port or state variable changed, the new value it will get, and the (future) simulated time at which it will attain that value. Asserting an event therefore involves passing this information to the simulator for later processing.

Ports are first-class citizens in SIMPLE, and events are asserted on them by means of the `assert-port` primitive. An output port can be retrieved via the `port?` function described earlier, and can thereby be passed as an argument to `assert-port`, along with its new value and the simulated time of the change.

State variables, on the other hand, are simply *places* (in the `setf` sense [13]) that may hold values. A state variable is therefore specified by the expression that will access the place that holds its value. Thus, for example, a slot denoting a top-level state variable of a component is simply specified by naming the slot. The `Status` slot of an operator is such a state variable. As a more general example, if `Registers` is a slot denoting a vector of simulated registers, then the expression `(svref Registers 0)` is an accessor for the state variable representing the first register. The `assert-state` primitive is used to generate an event on a local state variable. As with ports, there are no restrictions enforced by SIMPLE on the values held by state variables.

Processing Events

Once an event has been asserted, at the appropriate simulated time the simulator processes the event: that is, it makes the state change specified by the event and then invokes the behavior method that defines affected component's response to the event. The parameters

to the behavior method are a specifier for the port or state variable affected, its new value, and the simulated time of the change (which may be thought of as the 'current' time).

A behavior method is typically structured as a set of 'rules', and each rule tests for conditions and, as satisfied, asserts or directly effects consequent actions. Rule conditions may include arbitrary predicates on the event parameters as well as on the state variables and input ports of the component; predicates may be combined through Lisp operators such as `and` and `or`. Rule actions may directly manipulate the component's state as well as assert events in the simulated future.

`SIMPLE` supplies a number of primitive predicates for testing events. The simplest predicates test if the event occurred on a specified port or state. Others additionally test if the asserted value satisfies conditions such as equality with constants, membership in a set of values, or membership as defined by type.

Modelling Synchronous Designs

As discussed earlier, event-based simulators assume that state variables (including ports) remain unchanged until explicitly modified. Synchronous designs, that is, those in which the opportunities for state change are temporally quantized to a clock, can be modelled in such implicitly asynchronous simulators by asserting the clock signal on a port of each and every clocked component of the simulated system. However, if only *some* of the components in the system need take action on each clock signal (as is typical), there is an obvious inefficiency in this approach that is crippling for systems with even a modest number of components.

If, on the other hand, event times are restricted to integers, the clock can be assumed. All that is needed is a way to detect the event for which a Boolean combination of conditions as strobed by an assumed clock is *first* met. To support this, `SIMPLE` supplies primitive condition predicates for detecting an 'edge', that is, a value changed by the current event, with a coincident 'level', that is, a value set by an event in the simulated past, of two ports or state variables of a component in either of the two possible event sequences. The predicate `port-state?` in the example behavior rule shown in figure 5 has these semantics.

This code also illustrates the generality of `SIMPLE` behavioral descriptions. Actions may directly manipulate state variables (as is done to set `Status` to 'servicing'), assert events (as is done to the `Status` state variable and to the `Evaluator-Packet-Out` port), call arbitrary procedures (for example, `queue-take` and `time-update`), or call methods (such as `:operation-cycle`). In fact, the last approach has proven to be a natural way to realize the functional operations of `CARE` components not described by behavioral rules.

3 CARE Component Models

`CARE` defines a small number of multiprocessor components, both primitive and composite,

```

((and (port-state? Evaluator-Status-In 'free Status 'busy)
      (not (queue-empty Pending-Operations))
      (eq 'to-evaluator (operation-place (queue-top Pending-Operations)))))
;; If the operator is 'busy & there's something in the
;; queue for the evaluator & the channel to the evaluator
;; is 'free, then pop the queue and transmit.
(let* ((top (queue-take Pending-Operations)) : pop queue
      (post-time (send self :operation-cycle top now)) ; when
      (packet (operation-packet top))) ; what
  (time-update packet post-time) ; timestamp
  (setf Status 'servicing) ; block rule
  (assert-port Evaluator-Packet-Out packet post-time) ; xmit
  (assert-state Status 'busy (1+ post-time)))

```

Figure 5: A Behavior Rule

along with the data structures manipulated by them in support of the LAMINA concurrent language extensions. These are briefly described below.

3.1 Primitive Information Structures

The basic information structures manipulated by CARE components are the *process*, the *stream*, and the *packet*. Processes encapsulate a single thread of application code, and, perhaps, an address space. They communicate and synchronize by operating on streams, that may be thought of as queues that can store sequences of arbitrary values. Although streams are localized to a single processing site, they may be referenced by remote processes. Typical operations on streams involve treating them as message buffers, that is, sending and receiving messages on them, or treating them as memory cells, that is, reading and writing them. Operations on streams and processes are effected by packets of information being communicated between, and interpreted by, the components of the simulated multiprocessor system.

The LAMINA primitives can be used to model both shared variable and message passing styles of computation [6]. A LAMINA application program consists of sequential Lisp code interspersed with LAMINA language constructs that have been built from these primitives. During execution, the primitives cause events that result in control being passed to the simulator for their handling. In this way, a simulation achieves its goal of focussing on the interactions between processes.

3.2 Components

The component types supplied by CARE are essentially those shown in figure 2. These are elaborated upon below.

Communications Components

CARE supplies primitive communications components that accept (or block), route, and buffer transmissions in accordance with a dynamic, flow-controlled, cut-through communications protocol that includes support for multicast [4, 5]. Currently, these are the **net-input** and the **net-output**. All transmissions are encapsulated as packets containing routing and control information along with application data. To maintain integrity in the simulation, the data values transmitted in packets are copied before being passed to the communication subsystem, and packets are sized accordingly.

In keeping with the objective of focusing simulation cycles on the aspects of the simulation particularly relevant to multiprocessor operation, the behaviors of the communications components are defined in fair detail, that is, at approximately the register transfer level. Routing operations are described procedurally, as methods on the site or bus composite component enclosing the communications components. Parameters allow choice of the routing algorithm used, width of data channels, routing decision times and so on.

Processing and Scheduling Components

CARE supplies the **pme** (for 'processor memory element') to accomplish the processing work of a CARE system. This composite consists of an **evaluator**, an **operator** and at least two **fifo-buffers**.⁴ The storage associated with this component is not explicitly modelled.

The buffers interface the processing subsystem with the communications subsystem and are used for local packet receptions and transmissions. Their behavior is also described at approximately the register transfer level, and allows parametric control of buffer depth.

The evaluator does the real work of the application—running processes that execute application code. The operator does the overhead work associated with such evaluations, that is, managing processes and streams. In particular, the operator schedules processes for execution by its associated evaluator, receives and interprets request packets for operating on local streams (such as queueing messages on them), and constructs outgoing packets that require operations to be performed on remote streams and delivers them to the communications subsystem. Thus, depending upon the computation model, the operator can function as a message coprocessor or as a memory controller.

⁴There are specializations of the fifo-buffer component for input and for output.

As indicated previously, the simulation of the operator and evaluator has two aspects: the control of the flow of information and the actions performed on that information. The former is described in terms of SIMPLE behavior rules, register transfer by register transfer. The latter is described directly in terms of procedures, and the simulated time taken by such procedures is modelled. In the case of the operator, this is done as a function of the number of storage cells manipulated during the operator routine that handles some primitive operation. In the case of the evaluator, this is done as a function of the execution time on the underlying simulation vehicle. Care is taken to ensure that time due to such overheads as page faults and garbage collection are discounted in measuring application execution time on the simulation vehicle.

There are numerous parameters associated with operators and evaluators. These primarily affect performance and include quantities such as cycle times, interrupt times, process switch times, packet formatting times, and so on.

System Level Components

CARE systems consist of a number of sites interconnected in some regular topology. Sites may currently be embedded into *mesh*, *torus* and *bus* topologies. The basic site composite is parameterized to generate communications components for up to eight 'neighboring' sites; it also contains a local processor-memory element. Specializations of the site, for example, the *torus-site* and the *bus-site*, exist to fit the site into alternative topologies by supplementing the site routing procedures as appropriate to the topology.

3.3 Application Development

CARE has evolved to provide a number of features that aid in developing LAMINA applications.

- Full integration with the underlying Lisp program development tools such as inspectors, debuggers and editors. Components and the data structures they manipulate have abstraction interfaces that provide a summary of their state information when they are displayed in text form. These text abstractions are 'mouse sensitive' and so can be inspected at successively finer levels of detail if desired. Application and model code can be debugged via graphical inspection and manipulation of stack frames. Within the debugger, a single keystroke brings the relevant source code into the editor. Incremental recompilation allows changes to source code to take immediate effect, even within the interrupted stack frame. Thereupon, execution can be backed up and retried, given that intermediate side effecting code is safely re-executable.

- A means for running batch simulations via script files. The script files might contain commands that vary application-specific parameters and data sets, as well as system configurations and parameters—perhaps based on the results of runs previously completed. This facility has been used for performance experiments spanning several days.
- A means for *recording* simulation executions for later *replay* [7]. The only inherently non-deterministic quantities in a simulation run are those that capture the timing of sequential application code fragments on the underlying simulation vehicle. These timings are recorded into a file and may later be used to derive the deterministic behavior of the rest of the system, that is, replay the original run. This can be useful both for debugging and for varying instrumentation for the critical system simulation.

4 Building Instrumentation

The results of a simulation are primarily the insights it provides into the operation of the simulated system. Where the system designer may seek insights into system resource utilization or protocol deadlocks, the application programmer may seek insights that help to debug and understand the performance of a concurrent application program. With this in mind, the design for SIMPLE's instrumentation system was aimed at flexibility, while retaining efficiency to the greatest degree feasible.

4.1 Abstractions and Implementations

SIMPLE's instrumentation system is organized around *probe*, *panel*, and *instrument* abstractions. Probes *monitor* individual components, and, when appropriate, *supply* abstracted data they have collected to panels. Panels *transform* and *save* interesting data from particular kinds of probes in the system, *organize* the transformed quantities along various dimensions, and periodically *display* the results of summary *analyses* on this information. Instruments package together a collection of particular panels, thus providing simultaneous access to different views of operation of the instrumented system.

SIMPLE implements these abstractions by providing a library of classes, methods and procedures that obey a predefined *measurement protocol*. Probes, panels and instruments are built through instantiation of classes derived from the base classes, and the protocol provides the foundation for customizations that allow them to achieve the desired functionality. This is shown in figure 6.

SIMPLE is designed to make the specification of these customizations as incremental as possible so that existing solutions can be reused. The metaphor it provides to do this

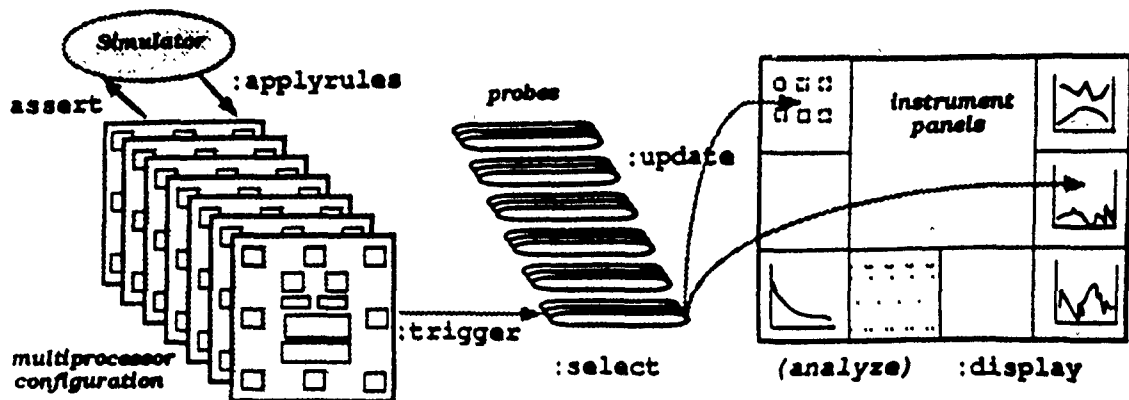


Figure 6: Instrumentation Runtime

is a familiar one: specialization, which is implemented through the extensive inheritance facilities of the underlying Flavors system.

Specializations may range from defining the body of a method invoked by the measurement protocol to providing default values for predefined slots that affect the behavior of the methods that implement the underlying protocol. Default slot values may, in turn, range from simple values such as strings denoting panel legends or functions defining probe filters, to lists representing code expressions that are parsed, compiled and called at run time to accomplish a panel's transformation, analysis and display operations. While a full discussion of the system-supplied opportunities for customization is beyond the scope of this article, the following sections will attempt to show that the design tries to ensure that simple things are simply specified.

4.2 Data Capture: Probes

Each probe is attached to a single component in the simulated design and is responsible for monitoring a particular aspect of its behavior. This monitoring is made non-intrusive by ensuring that a probe is informed of all events pertaining to its attached component. As shown in figure 6, the `:applyrules` method that defines a component's behavior also has a `daemon` method that invokes the `:trigger` method of all probes attached to the component, passing the event parameters to each. Each probe is then responsible for taking action based on the event, if desired.

Probe actions may involve filtering events, querying the values of ports and state variables on the attached component, manipulating the contents of the probe's own instance variables (thus, probes can be history-sensitive), and, finally, processing and forwarding data to attached panels. The processed data is formatted as a property list that tags each

datum with an identifying keyword symbol, and it is encapsulated with a *probed object* for which the data is relevant and a number representing the simulated time. The probed object can be an arbitrary data structure; it may be the attached component or one related to it (for example, the component enclosing it), a data structure manipulated by the component (for example, a process structure of an evaluator), or even an application data structure (for example, a LAMINA object).

An Example

As an example probe, consider the *evaluator-queue-probe* defined in figure 7. This probe measures the load on an evaluator in terms of the number of runnable (and running) processes queued on it. Since processes arriving from the local operator (via the *Packet-In* port) increment the load, and since transitions in the evaluator's *Status* reflect the status of the process currently being executed and thereby affect load, the *:trigger* method checks to see if the event on the attached evaluator is relevant before taking action. As with component behavior rules, this is done through the *state-event?* and *port-event?* predicates.

```
(defprobe EVALUATOR-QUEUE-PROBE () ; no mixins
  ;; instance variables: cache attached evaluator's slots
  ((Input-Queue (probe-state Evaluator-Queue))
   (Site (probe-state Site)))
  ;; options
  (:documentation "Report evaluator process queue lengths")
  (:component-type evaluator) ; attach to evaluators
  (:trigger (tag value now) ; name the event parameters
    (when (or (state-event? Status) ; status change?...
              (port-event? Packet-In)) ; or process arrival?
      (send self :select ; i.e. inform attached panels...
        Site ; probed object = site component
        (list :busy ; probe data property list
              (+ (queue-length Input-Queue) ; # enabled processes
                 (case (probe-state Status) ; running process?
                   ((ready stalled) 0) (t 1))))
              (simulated-microsecond-time now)))))) ; probe time
```

Figure 7: Example Probe Definition

The declarations of this probe's instance variables use the *probe-state* primitive to retrieve the values of slots in the evaluator and thus initialize the slots of the probe instance. In general, instance variables are used to store intermediate state as required for probes that

track interesting *sequences* of state changes (for example, the scheduling transitions of a process). Note that the site component that contains the evaluator is passed as the probed object for which the data is relevant. Note also the conversion of event time units (now) into model-specific time units, through scaling, as the data is passed on to panels.

The `:select` method is part of the measurement protocol for probes. The default method forwards the probe data on to attached panels through an system-supplied intermediary object. This object optionally filters the probe data and, if successful, tags it with an identifying keyword symbol as required by the panel (so that a panel may distinguish between different kinds of probes) before calling the `:update` methods of the selected panels. A probe filter is specified when defining (or instantiating) a panel that uses the required kind of probe.

4.3 Data Analysis and Presentation: Panels

Panel operations are accomplished by successive transformations on the data supplied by probes, ultimately yielding the quantities that are displayed along the various 'axes' defined by the presentation style of the panel. These transformations are conceptually accomplished through manipulations on two kinds of records:

- a *state record* for each probed object, that stores relevant information derived from the probe data passed in; and,
- a *display record* that stores the quantities that need to be displayed, and forms the foundation for display lists.

Display records are organized along panel-specific dimensions to satisfy display goals. These may be times, durations, frequency and counting bins, probed objects, and so forth. Both state and display records are created as needed by the panel.

The actions taken by a panel are then to:

- *update* the state and display records corresponding to the probe data passed as parameters to the `:update` call. This involves extracting the required data from the probe data property list, computing transformed values based on this and the retained data stored in the records, and storing results back into the appropriate records.
- *analyze* the display lists periodically, that is, reorganize them based on display objectives, such as sorting on display record fields.
- *display* the results of these periodic analyses in the display style of the panel, that is, transform display list quantities into graphics actions on the screen.

As mentioned earlier, the starting point for defining panels is *presentations*: class definitions that represent particular display styles. SIMPLE's current presentation library includes scrollable text displays, scatter plots, fixed and scrollable line plots, histograms, strip charts, intensity maps and signal animations. These are customized through specialization to define panels.

Customizations include those that affect the panel's graphical appearance, such as legends, scales, axes labels and the like, as well as those that achieve its functional objectives. The latter include declarations of the types of probes required to drive the panel, and *interface specifications*: arbitrarily complex expressions that specify the transformations between the information provided by the probes and that saved and displayed by the panel. Other customizations control the computing resources used by the panel; these are parameters such as sampling intervals, refresh periods, and history depths. Presentations have been defined so that they supply the most commonly required customizations implicitly.

To retain run time efficiency, the expressions provided in interface specifications are processed when a new instrument is created. They are compiled at that time into code bodies referenced by run time control blocks associated with the underlying methods that implement the panel measurement protocol.

An Example

As an illustrative panel definition, consider the code shown in figure 8. This defines a strip chart that plots the recent history of total evaluator queue lengths in the system over time, thus providing a view of the available application concurrency.

The important points about the specification are:

- The 'probes' specification, which declares the kinds of probes that are required and how the data they supply will be mapped into the transformation expressions that use the data. This specification uses a generalized binding format that pairs a keyword symbol—the *probe key*—with a particular kind of probe, which allows the panel to distinguish or combine data from different types of probes, as needed. Keeping probes isolated from the transformation expressions in this way allows different probes to be 'plugged in' to the panel by simply specifying a different binding list. The resolution from probe type to probe key is automatically performed by the simulation system as described earlier.
- The 'axis' specifications, which are expressions describing the transformations on probe data. Within an expression, the general form for denoting keyed data values supplied by a probe is as a list composed of the relevant probe key and the relevant data key, such as (:queue-probe :busy). These value expressions can be combined

```

(defpanel EVALUATOR-QUEUE-HISTORY-PANEL
  ;; slots and initializations
  ((name "EVALUATOR QUEUE HISTORY")
   (legend "Total Evaluator Queue Lengths")
   (time-scale-factor 0.001) ; us [from probes] to ms [display]
   (sampling-interval 200) ; 1 sample kept per 200us
   (scroll-range 10) ; 10ms 'window' of time displayed
  ;; interface specifications
  (probes '(:queue-probe evaluator-queue-probe)))
  (left-axis-form '(:queue-probe :busy save-sum)) ; queue lengths
  (bottom-axis-form '(:simulator :time)) ; reported probe times
  (plot-update-form '(send self :update-time (:simulator :time))))
  ;; slots that are reset between simulation runs
  ((left-axis (make-axis :label "Evaluator Queue Sum"
                        :range (make-range 0.0 nil))) ; open ended
   (bottom-axis (make-axis :label (format nil "MS by "DUS"
                                         sampling-interval)
                          :range scroll-range))) ; fixed range
  ;; inheritance -- base SIMPLE presentation class
  (scrolling-line-plot-presentation))

```

Figure 8: Example Panel Definition

with others as required (through built-in or user-defined functions) to compute derived values. For example, one definition of 'load' on a resource in CARE is through the formula $1 - (1/1 + Q)$, where Q denotes the total lengths of all the queues that need to be serviced by that resource.. Its corresponding transformation expression might be

```
(- 1.0 (/ 1.0 (1+ (:queue-probe :busy))))
```

The optional *save-sum* modifier in the probe value expression for the 'left axis' introduces a summation transformation, which requires that the overall sum be decremented by the previous *:busy* value reported for the probed object and then be incremented by the new reported value. Were the modifier absent, the relevant display record would simply reflect the latest value reported; instead, it now maintains the running total of the latest reported values per probed object. SIMPLE has a number of such *save functions* to aggregate and classify data for display; it also provides a means for new ones to be defined.

- The 'update form' specification which ensures that the panel organizes display lists

along the dimension of simulated time, corresponding to the 'bottom axis' of the display. In general, this needs to be specified only when mapping time; otherwise, the default update behavior is sufficient.

This panel does not need the analysis feature that most panels provide as an option. SIMPLE's basic analysis operation allows sorting display lists by arbitrary predicates applied to arbitrary record fields. This is expressed through an 'analysis form' declaration such as

```
(sort-arrays  
  (list (list #'> (:latency-probe (+ :launch :network))))))
```

This code specifies that display records are sorted in decreasing order of the sum of the 'launch' and 'network' delays reported by a 'latency' probe (presumably monitoring communication latencies). The list of lists format of the specification allows for progressively finer sorts on items that are equivalent with respect to a coarser sort predicate.

5 Understanding Instrumentation in CARE

In this section, we will try to show how instrumentation helps understand the operation of concurrent CARE systems. To do this, we will focus on a particular programming model—the LAMINA object-oriented model, and its corresponding multiprocessor model—a message-passing CARE multicomputer.

5.1 Monitoring Computations

An object-oriented LAMINA application consists of objects that interact only by asynchronously passing messages containing data values [6]. Objects execute the messages arriving on their local task streams serially. Each message execution, or task, atomically manipulates the message contents and the object state and then sends new messages, thus continuing the computation at some other object.

The CARE message-passing machine model provides the resources that accomplish the LAMINA computations described above. Evaluators run the processes that execute the LAMINA object tasks. When a task needs to send a message, the evaluator interrupts the local operator and passes it the message data. The operator encodes the data values into a packet and passes it to the communications components for remote delivery. These route and deliver the packet to the remote site according to some communications protocol. The operator at the target site queues the message packet on the relevant task stream and perhaps reschedules a waiting object process for execution in the local evaluator.

A LAMINA application can thus be effectively monitored by simply monitoring the critical operations performed on messages by LAMINA objects, namely, the *generation* of messages,

the *arrival* of messages on the target object's task stream, and the *execution* of messages. The performance of the application can then be understood by monitoring the actions performed by the underlying system resources in supporting this message traffic, namely, the *creation*, *communication* and *receipt* of packets, and the *scheduling* and *execution* of processes.

This captured information provides a basis for understanding system operation. The impact of the application decomposition can be studied in terms of task and message granularities, message volumes and frequencies, over- and under-utilized objects and classes, and so on. The impact of the system design, its operating parameters, and its finite resources can be studied in terms of resource utilization, service latencies, resource conflicts, load imbalances, resource bottlenecks and so on. Some examples of how these may be understood using CARE's instrumentation are given in the next section.

5.2 Seeing System Activity

In this section, we will describe some representative panels to illustrate the visualization capabilities of CARE's instrumentation.

Activity and Load Maps

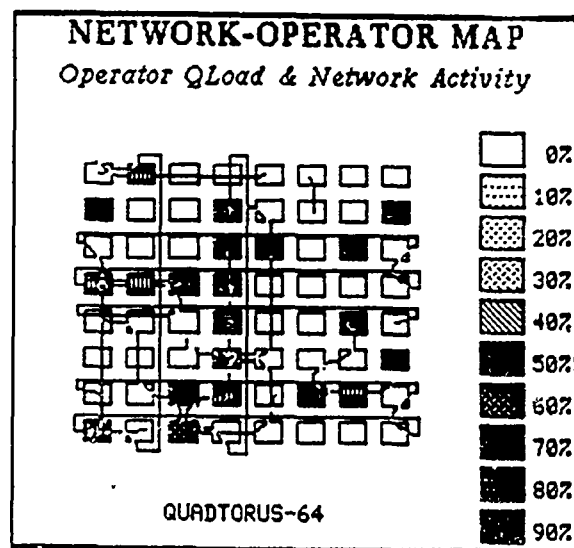


Figure 9: Mapping Panel

One of the most intuitive kinds of presentations is the *mapping* presentation. It provides an animation of activity in the design in terms of the spatial arrangement the system designer

laid out when the structural organization of the design was defined. The 'Network Operator Map' panel shown in figure 9 uses this topology to display loads on the operator resources in the system and the activity in the interconnection network.

The boxes in figure 9 correspond to operators in the system. Their shading indicates how many packets are queued up for service by the corresponding operator, using the formula for 'load' given on page 21. The indicated load on operators shifts as the simulation proceeds so that bottlenecks in operator resources stand out visually. Load imbalances show up as more or less constant utilization of only certain operators.

The lines between boxes correspond to connections made for packet transmission between the network ports of neighboring sites, so that a qualitative view of the degree to which the network is utilized at a given time in the simulation is available. We have found this useful in debugging the network protocols that we have experimented with—deadlocks and thrashing are often immediately apparent.

In CARE, mapping presentations have been specialized to create a number of different panels. We have found it useful for seeing object load (the number of LAMINA objects at a site), message load (the sum of the lengths of all LAMINA task queues at the site), evaluator load (the number of runnable processes at a site), or, simply, evaluator status.

Utilization Histograms

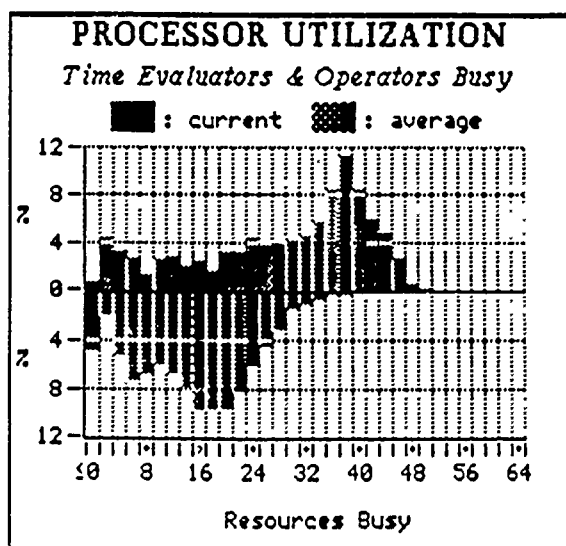


Figure 10: Utilization Histogram

A more statistical view of the operation of the system over time is provided by the depiction of, for example, the utilization of operator and evaluator resources as *histograms*. The 'Processor Utilization' panel shown in figure 10 shows the percentage of time that a given number of evaluators, displayed on the top half of the panel, and a given number of operators, displayed on the bottom half of the panel, have been used. Highlighting shows what the current situation is (37-38 evaluators are busy) as well as what the average situation has been through the current time of the simulation (27-28 evaluators have been busy simultaneously on average).

If this panel indicates that only a few evaluators are concurrently active, it may be either that the application is not generating enough concurrency, or that the processing load is unevenly balanced so that the potential concurrency is not being exploited. Other panels, described below, may be used to clarify this explanation.

Load and Latency Strip Charts

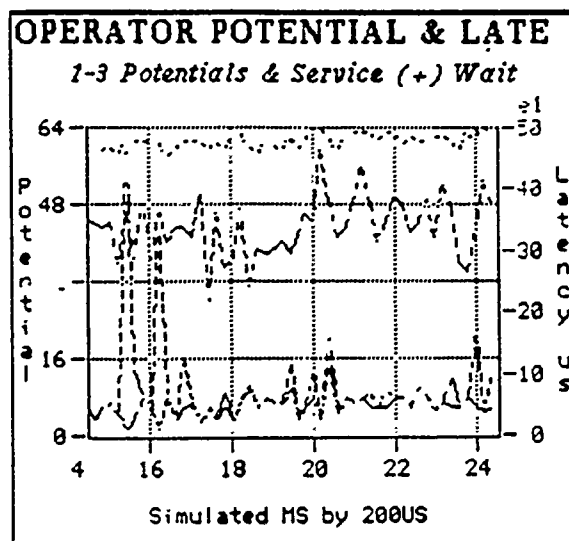


Figure 11: Activity Stripchart

A *strip chart* presentation is a useful way to see what the history of some measure of system activity has been in the recent past. There are four such measures plotted in the 'Operator Potential & Latency' panel shown in figure 11.

Two of the measures are plotted on the scale at the right side of the strip chart, and they show the latency being experienced by operators in the system as they receive and service packets from the network. The time to service packets is the lower of the two plots,

and it ranges from 50 to 100 microseconds in this case. There are occasional delays between the time a packet is received and service on it is begun. This delay is shown as an offset on top of the time plotted to service packets.

If the latencies shown by this panel have a periodic character with unacceptable peak times, it may be that there are load imbalances that can be addressed to improve this situation. Alternatively, more or less monotonically increasing latencies indicate application pipelines that are not keeping up with their inputs. If the affected pipelines can be replicated and work spread among them, or if the grain size of the larger pipeline stages can be reduced—and the resources are adequate to the demand—the bottlenecks causing the increasing latencies may be broken.

The two upper measures plotted in Figure 11 refer to the 'potential' for additional work remaining in the system. Their scale, indicating the number of idle operators, is shown on the left side of the strip chart. The lower of the two plots indicates the number of operators that have no packets in their service queues. The remaining measure plotted is similar, and it indicates the number of operators that have less than three packets in their service queues.

The values of these potential plots is an indication of resource utilization over time. The distance between them is an indication of load balance: if they are well spread, most of the operator resources have one to three packets to handle, which is an indication of good load balance. Alternatively, both plots close together toward the middle of the axis indicate that half of the resources described have more than three packets to handle and half have none: an indication of poor load balance. Both plots drawn down toward the bottom of the panel may indicate an overloaded system: all the resources being monitored have several packets in their service queues.

As shown in figure 14, similar panels have been defined for the communications subsystem (the 'Network Load & Latency' panel) and the processing subsystem (the 'Evaluator Potential & Latency' panel). These can be used along with the one described here to see the relative granularity among the subsystems and their relative utilization so as to discover the critical resources in the system.

Activity Tables

Often the most informative way to present data is as text tables. The two panels shown in figure 12 use *scrolling text* presentations to dynamically summarize the activity of LAMINA objects in the system.

In the 'Activity By Instance' panel, each line in the scrolling display represents a single LAMINA object in the application. The columns of the tables denote, respectively:

- the expected service time of the object, that is, the product of its average task execution time and the number of messages in its task stream. This is an indication of

ACTIVITY BY INSTANCE					ACTIVITY BY CLASS				
Service/Q Avg (Runs) Delay Site					Service/Q Avg (Runs) Instances				
1-3/11	0-115 (34)	1-446	(7 5)	WETTER-OBSERVATION 12-0	0-2/ 0-6	0-298 (135)	15	(CLUSTER-STATUS - CLUSTER-REFOR	
1-2/ 4	0-308 (18)	2-302	(7 8)	WETTER-FIX 5 (7 8) 16902	0-1/ 1-0	0-137 (29)	3	(CLUSTER-TIMER - TIME)	
1-0/ 4	0-239 (24)	0-727	(7 4)	WCLUSTER-STATUS 3 (7 4) 13	0-1/ 1-0	0-104 (39)	3	(CLUSTER-TIMER - WATCH)	
0-8/ 6	0-153 (11)	0-215	(4 2)	WCLUSTER-TIMER 5-0 (4 2) 2	0-1/ 0-2	0-320 (305)	20	(WETTER-FIX - REPORT)	
0-8/ 3	0-250 (25)	0-306	(7 5)	WCLUSTER-STATUS 3 (7 5) 13	0-0/ 0-6	0-081 (62)	20	(WETTER-OBSERVATION - UPDATE-0	
0-7/ 3	0-246 (26)	0-230	(7 8)	WCLUSTER-STATUS 3 (7 8) 13	0-0/ 0-2	0-201 (312)	20	(WETTER-OBSERVATION - REPORT)	
0-5/ 2	0-289 (26)	0-313	(7 7)	WCLUSTER-STATUS 2 (7 7) 13	0-0/ 0-1	0-223 (171)	10	(CLUSTER-STATUS - WATCH)	
0-4/ 3	0-118 (62)	0-198	(7 1)	WETTER-OBSERVATION 4-0 (0-0/ 0-0	0-588 (1)	1	(ELINE 0)	
0-3/ 1	0-260 (28)	0-298	(7 6)	WCLUSTER-STATUS 0 (7 6) 13	0-0/ 0-0	0-063 (1)	1	(ELINE 0 0)	
0-2/ 1	0-234 (35)	0-227	(4 2)	WCLUSTER-STATUS 2 (4 2) 72	0-0/ 0-0	0-201 (3)	1	(CLUSTER-MANAGER - CONTINUOUS)	
0-1/ 1	0-116 (29)	0-110	(5 3)	WETTER-OBSERVATION 0-2 (0-0/ 0-0	0-242 (3)	3	(CLUSTER-TIMER - SET-CLUSTER-M	
0-0/ 0	0-141 (17)	1-180	(7 6)	WETTER-STATUS 1-0 (7 6)	0-0/ 0-0	0-169 (3)	3	(CLUSTER-TIMER - SET-CLUSTER-PL	

Figure 12: Activity Tables

the degree to which this object is a bottleneck. The text lines are periodically sorted so that the objects that have the highest expected service time bubble to the top of the display. Objects forming potential bottlenecks are thereby evident.

- the number of messages on the object's task stream.
- the average task execution time for the object, that is, the average time it has taken to process a message up to this point in the simulation.
- the number of messages that have been processed by the object, an indication of its relative activity.
- the delay experienced by the most recent message that was executed (as a task) by the object. (More precisely, the panel reflects the situation when it was last refreshed.) This delay is the interval from the generation of the message by the sending object at some remote site to the actual execution of the task by this object at its site. It represents the overhead involved in getting the task accomplished, and, as such, includes the latency in getting the message delivered as well as the scheduling delay before the task corresponding to the message is executed, which may include multiple schedulings of the process corresponding to the object.
- the site at which the object is located. This can be used to discover if the object is bottlenecking because of load imbalance, and this is apparent if the most backed-up

objects are colocated.

- a printed representation of the object, showing in particular its class. Text lines are 'mouse sensitive' so that the LAMINA object can be inspected through a simple mouse click.

The 'Activity By Class' panel presents this information aggregated by the class of object and type of message, as shown in the rightmost column of the display. This information can be used to see the distribution of work due to the application design. An inappropriate distribution may indicate the application needs to be reorganized; the display provides guidance about where this effort should be concentrated.

Cumulative Latencies

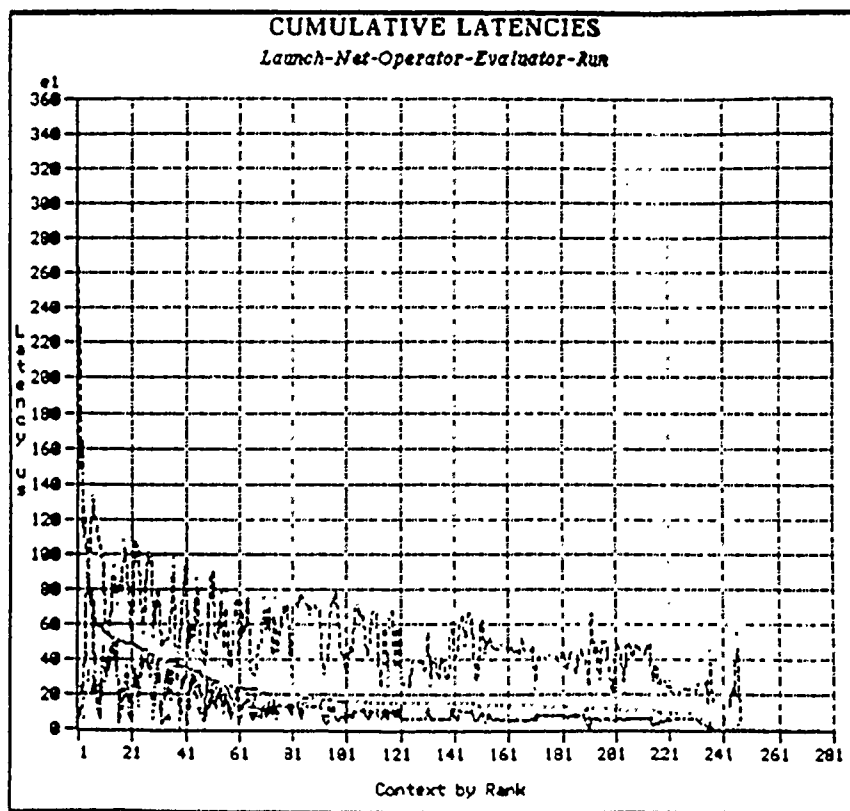


Figure 13: Cumulative Latencies

The 'Cumulative Latencies' panel in figure 13 is an example of a *line plot* presentation. It displays a snapshot of the same message delays described above, but as experienced by the most recent messages received and processed by each of the extant application objects. There are five curves, incrementally showing the latency experienced by the messages at the source operator, being routed in the network, waiting for service at the target operator, being serviced by the target operator, waiting for execution at the target evaluator, and, finally, being executed as the task that consumes the message. The curves are ranked by the sum of the first four delays above, which represents the overhead in getting the requested task accomplished at the targeted object.

5.3 A Complete Instrument

The panels described above have been collected into the CARE 'Observer' instrument shown in figure 14. Additionally, the instrument provides an *annotation* panel reflecting system parameters and other data, so that experimental parameters are evident.

The instrument thereby provides a unified view of system operation that correlates the activity of hardware abstractions, that is multiprocessor subsystems, with application abstractions, that is, LAMINA objects.

6 Conclusions

The SIMPLE/CARE effort began on the premise that multiprocessor systems could be studied through simulation at an architectural level while being driven by significant concurrent applications. SIMPLE/CARE has evolved to support this thesis, by coupling flexibility with reasonable performance. The key features that enable it to do so are:

- Component-based design capture that is tightly coupled with the underlying object-oriented programming system, and which allows systems to be built up incrementally and hierarchically.
- Arbitrary data types and lengths in simulation. The information whose creation and flow is controlled by simulated components may be of arbitrary complexity—from integers and symbols to procedure bodies and execution environments.
- A modular instrumentation architecture that permits system instrumentation to be modified independently of the system design. The architecture offers a broad range of customizations, including: probes for data capture and panels for data presentation, arbitrary expressions for probe data transforms, many-to-many probe to panel mappings, summary analyses on data, and a variety of display styles. SIMPLE's current library of display styles includes sorted, scrollable text lines as well as self and

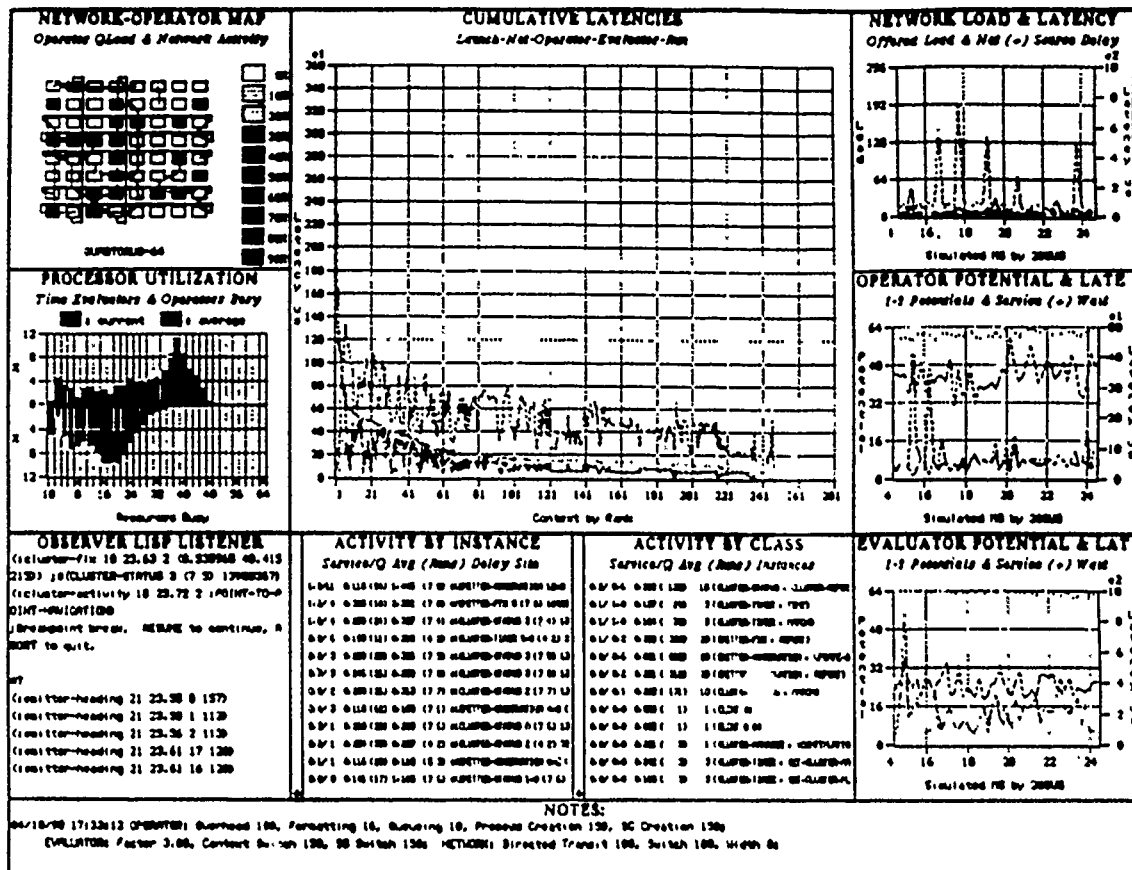


Figure 14: CARE 'Observer' Instrument

fixed scaling, 'two and a half' dimensioned, history-sensitive displays that may be line graphs, histograms, strip charts, intensity maps and signal animations.

- Leverage from the underlying Lisp environment that provides a comprehensive suite of tools for program development (such as debuggers, inspectors and method dictionaries), allows changes in model or application definitions to have instantaneous effect, and provides quick access to source code.
- A focus on the interactions between multiprocessor system components, which improves performance without sacrificing critical detail. The CARE component library simulates the behavior of network components at the register transfer level while directly executing and timing purely sequential application code.
- An application language interface that is easily modified without recasting the information flow control defined by CARE component behavior.

Researchers within the Knowledge Systems Laboratory have used SIMPLE/CARE to study a broad spectrum of problems in concurrent systems [9]. Their experiments have involved simulation runs of a few minutes to many days in duration. While faster would surely be better, performance has for the most part proven adequate to their needs.

Acknowledgements

This work stands on the shoulders of its predecessor, the PALLADIO system, designed and implemented by Harold Brown and Gordon Foyster. Our functional goals were more restrictive than theirs so we had the luxury of design by simplification. Without their implementation base, it would have been hard to know even where to begin.

Many hands and minds have contributed to the development of SIMPLE/CARE. We are particularly indebted to Greg Byrd, Max Hailperin, Russ Nakano, and James Rice, all of whom contributed significantly to both the design and the implementation of the system. We are grateful to the members of the Advanced Architectures Project for their patience as users of the developing (and undocumented) system.

Finally, we would like to thank the staff of the Symbolic Systems Resources Group of the Knowledge Systems Laboratory for providing excellent support of our computing environment. Special thanks to Ed Feigenbaum for his work in support of the Knowledge Systems Laboratory and the Advanced Architectures Project.

References

- [1] Nello Aiello. The Cage system users manual. Technical Report KSL-89-86, Knowledge Systems Laboratory, Stanford University, 1989.
- [2] John Batali and Anne Hartheimer. The Design Procedure Language manual. A.I. Memo No. 598, Artificial Intelligence Laboratory, M.I.T., September 1980.
- [3] Harold Brown, Christopher Tong, and Gordon Foyster. Palladio: An exploratory environment for circuit design. *Computer*, 16(12):41-58, December 1983.
- [4] Gregory T. Byrd, Russell Nakano, and Bruce A. Delagi. A dynamic, cut-through communications protocol with multicast. Technical Report STAN-CS-87-1128, Department of Computer Science, Stanford University, September 1987.
- [5] Gregory T. Byrd, Nakul P. Saraiya, and Bruce A. Delagi. Multicast communication in multiprocessor systems. In *International Conference on Parallel Processing*, 1989. Also available as KSL-88-81, Knowledge Systems Laboratory, Stanford University.
- [6] Bruce A. Delagi, Nakul P. Saraiya, and Gregory T. Byrd. LAMINA: CARE applications interface. In *Third International Supercomputing Conference*. Information Sciences Institute, 1988. Also available as KSL-86-67, Knowledge Systems Laboratory, Stanford University.
- [7] Max Hailperin. Instant replay for simple, care. Private communication via electronic mail, September 1988.
- [8] P. Kermani and L. Kleinrock. Virtual cut-through: A new computer communication switching technique. *Computer Networks*, 3:267, 1979.
- [9] James P. Rice. The Advanced Architectures Project. *AI Magazine*, 10(4):27-39, Winter 1989.
- [10] James P. Rice. The design and implementation of Poligon, a high-performance, concurrent blackboard system shell. Technical Report STAN-CS-1294, Computer Science Department, Stanford University, 1989.
- [11] Nakul P. Saraiya. A shared-memory Lisp package for CARE. Technical Report KSL-88-85, Knowledge Systems Laboratory, Stanford University, January 1989.
- [12] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79-109, April 1986.

- [13] Guy L. Steele Jr. *Common Lisp: The language*. Digital Press, 1984.
- [14] Symbolics, Inc. *Lisp Machine Manual*, 1985.
- [15] Daniel Weinreb and David Moon. *Flavors: Message-passing in the Lisp Machine*. A.I. Memo No. 602, Artificial Intelligence Laboratory, M.I.T., 1980.

The LAMINA Programming Model: A Worked Example

by
Nakul P. Saraiya
and
James P. Rice

**Knowledge Systems Laboratory
Stanford University
701 Welch Road
Palo Alto, CA 94304**

The authors gratefully acknowledge the support of the following funding agencies for this project; DARPA/RADC, under contract F30602-85-C-0012; NASA, under contract number NCC 2-220; Boeing Computer Services, under contract number W-266875.

Abstract

Lamina is an experimental programming framework that allows concurrent algorithms to be expressed using both value-oriented and reference-oriented styles. It provides mechanisms and syntax (as extensions to Common Lisp [Steele 84]) to describe and control concurrent computations so that their performance may be studied using the Simple/CARE architectural simulator [Delagi 88]. This paper describes the Lamina *functional, object oriented message passing* and *shared variable* programming models, along with a simple worked example of their use. It also describes the underlying primitive operations that support the models.

1. Introduction

In this paper we present Lamina, a programming model developed as part of the Advanced Architectures Project at Stanford University's Knowledge Systems Laboratory [Rice 88]. To motivate the value of the programming constructs introduced we use a trivial worked example, involving simple operations being performed to matrices. This worked example is implemented in a number of different ways, using Lamina's Functional Programming, Shared Variable and Object Oriented programming constructs.

Although the Lamina language is primarily a research vehicle, it should be noted that it has been used to address non-trivial programming tasks. Large, expert system applications have been implemented using Lamina as part of our research. The utility of the programming constructs described herein is therefore well established.

This paper is a rewritten version of a previously published technical report, KSL-86-67. It does not present any materially different ideas, it does, however, present the ideas of the Lamina programming model with the use of a new, simpler worked example problem. This should help to focus the reader's attention on the broader significance of the programming model rather than specific implementation details. In fact, the worked examples, although written in Lamina's extended form of Common Lisp, were written in a manner that minimized the use of Lisp-specific syntax or constructs. Thus a reader who is unfamiliar with Lisp should still be able to understand these worked examples in terms of more common programming languages, such as C.

1.1. Cells, Futures and Streams

Cells form the basis for perhaps the simplest form of interprocess communication and synchronization. Communication is accomplished by reading and writing cells that are shared between concurrent processes. Synchronization can be accomplished by having the memory system support an atomic *read-modify-write* cell operation (such as an exchange).

Futures [Friedman 80] and [Halstead 85] can be thought of as cells that represent promises for potentially unavailable values. They can be used as placeholders in a computation while their values are being *eagerly* [Lieberman 81] produced by concurrent evaluations for consumption as available. Futures therefore embody both communication (of the produced value) as well as synchronization (because the value must be produced before it can be consumed). *Streams* [Lieberman 81] and [Shapiro 86] generalize futures by representing *sequences* of eagerly produced but potentially unavailable values as a single abstract data type. Streams can thus be used to build pipelines of computation connecting the producers and consumers of values.

Streams and futures may be the arguments to or the results of function applications. Furthermore, certain operators (sometimes called *non-strict* operators) do not require the actual values promised by a stream or future in order to perform their work. For example, a constructor (such as `cons`) may create data structures that include streams as elements without accessing any of the promised values the streams represent; referencing the placeholders is sufficient.

Lamina provides the `stream` as its primitive data type; a `future` is a specialization of a stream that represents only a single value. Streams and futures, because they represent arbitrary values such as lists and vectors, must be managed by a resource such as a processor—with attendant costs. Cells, however, can hold only single, fixed-size quantities such as small integers or references to other cells; thus, operations on cells (such as `read` and `write`) can be efficiently handled by simple memory controllers.

1.2. Multi-Level Address Spaces

Lamina's address space design is based upon expectations about the expenses involved in global storage reclamation. If references (pointers) are allowed to exist between processor address spaces, relocation of the referenced data (for example, as required by a copying garbage collector) requires global synchronization, which can be expensive. Lamina's multi-level addressing scheme therefore creates inter-processor references only as necessary, so as to allow for independent, globally unsynchronized storage reclamation to the greatest extent possible.

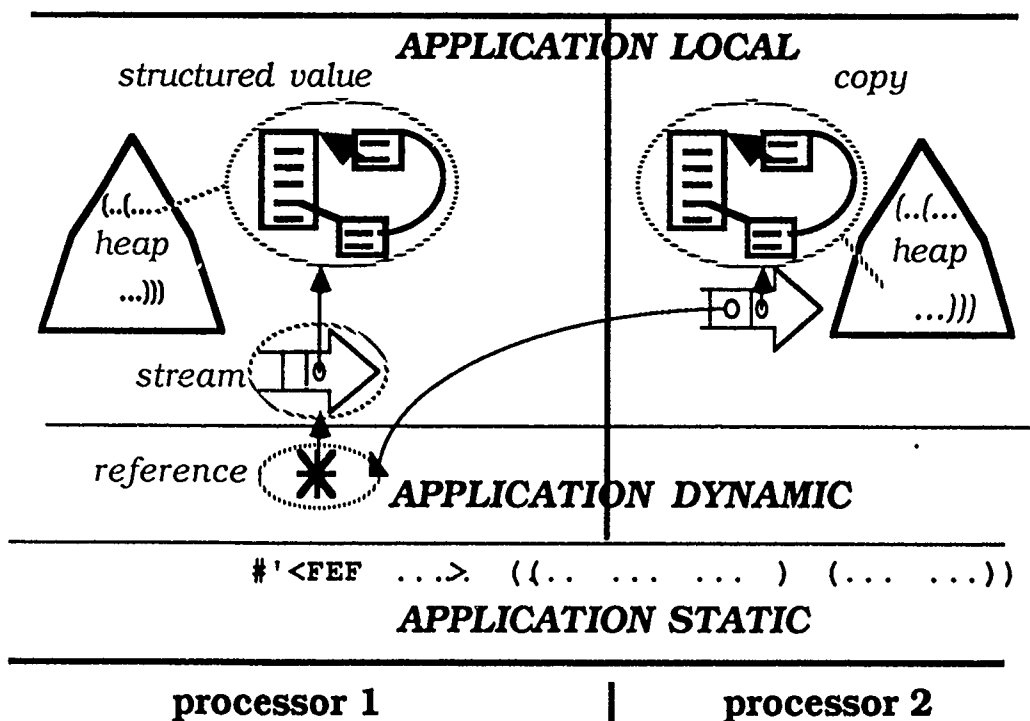


Figure 1. Local, Dynamic, and Static Addresses

As shown in Figure 1, an application's address space consists of

- *static space* containing data structures such as code bodies and constants (e.g., keyword symbols), which are regarded as immutable. They are therefore neither relocatable nor reclaimable, and so may be freely referenced and cached by any

processor. Lamina does not explicitly model transactions concerning data in static space; it assumes that static data is always available in a processor's cache.

- *dynamic space* containing cells (in the shared variable model), and *indirect references* to streams and futures. Indirect references may be thought of as remotely unreadable and unwritable 'reference cells' containing pointers to local data structures that represent streams and futures. References to data structures in dynamic space are allowed to exist between processor address spaces; hence, the data structures may only be relocated through globally synchronized operations affecting all computations that could access them. Note, however, that streams and futures (and the data values that they represent) may be locally and asynchronously relocated because of the indirection involved when they are remotely referenced.

Streams, futures and cells are *only* visible as references in Lamina. In the remainder of this discussion, then, the terms 'stream', 'future' and 'cell' should be taken to be equivalent to references (perhaps indirect) to data structures of the appropriate type.

- *local space* containing arbitrary local data values. Local data structures cannot be remotely referenced and are always copied between processor address spaces. They may therefore be independently reclaimed and relocated.

1.3. Communicating Values

In Lamina, a data structure of arbitrary complexity can be supplied as a value of a stream or future either local or remote to the processor address space in which the structure was generated. This is passed by copying, so that the structure is isomorphically reproduced at the target stream or future.

When values are passed between processor address spaces, the structure representing the value, that is, the *structure value*, is recursively encoded until a data structure is produced which has the same form and internal relationships as the original value but which holds only: *static references* to structures in static space, *dynamic references* to structures in dynamic space, *internal references* to elements of the new structure value, and *self-referentials* or 'immediate' data objects such as small numbers. This encoded data structure thus contains all the information required to form a copy of the original structure at the target stream or future, through the reverse operation of decoding.

Depending on the underlying system, encoding of a structure value might be done asynchronously with evaluation of the user application, so if changes are to be made (at any depth) in the structure passed between address spaces, independent copies of the structure should be formed.

An example of values and references passed between processor address spaces is shown in Figure 1. One of the values of the stream in the application's processor 2 local address space is an independent copy of the structure value in the application's processor 1 local address space. Both structure values are heap allocated from independently managed heaps in separate local spaces. The other value shown for the same stream is an indirect reference to the other stream; the stream, in turn, represents (or *contains*) the original structure value.

1.4. Storage Management

The cheapest approach to the dynamic allocation (and deallocation) of memory is *stack-based* and local. However, the benefits of stack-based operation come at the cost of a

prescribed order of deallocation. Additionally (at least for the commonly used memory management enforced stack limit schemes) stack-based operation entails a minimum storage commitment that is significantly larger than the rest of the execution environment for each small granularity evaluation expected for Lamina programs. Stack based allocation can be used whenever references to structures with dynamic extent [Steele 84] are known to be entirely within a given sequential computation.

The next cheapest approach, for references that are local with indefinite extent [Steele 84], is heap based allocation in *local* space. Since such references are confined to a single processor address space, their referents may be allocated, relocated, and reclaimed asynchronously with operations on other processors and memories, based on just the information in the associated processor address space.

Finally, as the most expensive approach, global references may be made to dynamically allocated references (that is, to cells and reference cells) which must be relocated under a global synchronization scheme. Allocation in dynamic space is done independently by each processor and each allocation is distinct. Operations involving dynamically allocated references are handled by the processor (or memory controller) associated with the reference. The referents for such references (that is, the streams and futures) are mutable, and may be viewed as uncacheable.

References to locally allocated structures can also be passed between processor address spaces, by encapsulating them in streams and then passing out the (indirect) reference to the stream. By this indirection, pointers to locally allocated structures are held locally (and may readily be relocated) but a means is provided to reference them in other processor address spaces.

2. LAMINA Primitives

In this section we discuss the Lamina programming language primitives.

2.1. Creating Streams

Streams are created by the primitive function `new-stream`, which returns a reference to a new stream on the executing (that is, local) site. Futures—streams that have at most one value—may be created by the function `new-future`. Streams and futures may be labelled for debugging purposes by including a 'tag' as the optional first argument of its constructor, as in

```
(new-stream 'requests)
```

The default is for a stream to inherit a tag identifying the execution which creates it.

As described earlier, streams and futures are only visible as references. The *site* of a reference, that is, the processor on which it was created, may be determined by executing

```
(reference-site reference)
```

which returns a *site identifier* that may be used to specify sites as required for parameters to other Lamina primitives. References can also be tested to determine whether they are the 'equal' by the function `eq-reference`, a predicate that tests if the two supplied references are to the same, potentially remote, stream or future.

A stream may be thought of as an ordered queue of postings, each containing, among other things, a value. The default order of postings on a stream is non-deterministic arrival order. Sometimes, however, it is desirable to override this default so as to control the order in which values are consumed from the stream.

A stream ordered by increasing numeric keys, supplied as part of the postings it receives, can be created by the function, `ordered-stream`; this is typically used to prioritize the values currently available on the stream. Similarly, a stream that provides values in strict sequence according to non-decreasing integer keys, again supplied as part of the received postings, can be created via `sequenced-stream`; this is typically used to minimize scheduling overhead by deferring executions involving the consumption of 'out of order' values. Both these kinds of stream have application in the Lamina object oriented programming model discussed in Section 5.

When a locally allocated data structure needs to be passed between potentially concurrent computations as a reference rather than as (a copy of) its value, the form (`reference item`) returns a reference for the value of the item. This is implemented by placing the value on a local *valued stream* which can then be remotely referenced.

2.2. Producing Values for Streams

Streams acquire values as a result of postings received by them. This is directly done by a producer using the posting operation as in

```
(posting value to targets ...)
```

The operation is *non-blocking*; it immediately returns and the actual transmission of the (copied) value will occur some time later.

The posting may be multicast [Byrd 87] by supplying a list of target streams rather than a single target, so that each will receive a unique copy of the value. Additionally, there is a facility for *specializing* the value transmitted in a multicast to the individual targets of the posting. Any place a stream is used as a target of a posting, it may be replaced by a cons of that stream and the value specialization for that stream. The value specialization will be prepended to the supplied value and the combined list will be taken as the value of the posting when it arrives at the target stream. Specialization is specified by a list of lists even if only one target is involved, in order to distinguish it from a list of unspecialized targets.

The keys required for correct operation of ordered and sequenced streams can be included in postings by specifying a number following the keyword 'by' in the call creating the posting. Other keywords are also available, and, since they are used by many of the Lamina primitives, they are listed here.

- `to`, on *targets*: A target stream or list of targets streams for the indicated primitive Lamina operation. Some primitives expect site targets rather than stream targets, as discussed in later; for these, if no site is provided and one is needed, an unspecified site is chosen. The choice between the alternative keywords shown is purely stylistic.
- `for clients`: A stream or list of streams acting as the continuation of the computation that will be triggered by the Lamina operation.
- `as tag`: Arbitrary data for debugging. Defaults to the tag of the sending execution.

- *by order-key*: A number which may be used to order information in target streams.
- *after delay*: A positive number indicating the number of milliseconds that the operation will be delayed before being attempted.
- *with properties*: Arbitrary data intended for user extensions of the posting protocol.

2.3. Consuming Streams

The primitive *first-posting* returns the first posting of those present on the referenced stream. The primitive *next-posting* does the same but also removes the posting from the stream. Finally, *last-posting* returns the last posting and eliminates all others on the stream.

If the stream is empty, the three stream posting access functions return *nil*. Otherwise, they return a posting as a list consisting of the *value*, *clients*, *key*, *tag*, *origin*, and *properties* of the posting. For convenience, these elements of this list may also be accessed by the *posting-* primitives: *-value*, *-clients*, *-key*, *-tag*, *-origin*, and *-properties*. The number of postings available on a referenced stream is returned by the primitive *postings*.

If it is desired that execution be blocked until there is a posting for a specified stream, the stream posting access forms above may be wrapped in an *accept* construct, as in

```
(accept (next-posting stream))
```

In this case, when a posting is available on the indicated stream, the posting is returned to the restarted or resumed execution.

Futures in Lamina are defined so that their value, once attained, cannot be removed. Hence only the *first-posting* operator is a valid accessor for a future.

The access primitives described above will, if necessary, coerce the referenced stream into one local to the calling site (through *relocating* as described later). Sometimes, this is not the desired behavior, so a way is provided to access potentially remote streams without incurring this side effect.

2.4. Remote Streams

Posting-by-posting access of the information on streams may also be accomplished by requesting that a stream access function be applied to the streams at the site they exist on, as in

```
(accessing access-function on targets for clients ...)
```

The *access-function* may be any of the stream posting access functions, for example, the function *next-posting* described previously. A posting will be sent to the client streams when one is available on a target stream. This is the only way provided for expressing competitive access to a common stream.

An interlocked operation on streams is provided by

```
(exchanging value on targets for clients ...)
```

This causes last-posting to be applied to each target stream and the result sent to each client stream. The *value* replaces the last posting on the target stream. The exchange is atomic with respect to each stream.

2.5. Managing Streams

Streams in Lamina may be managed in various ways across the system.

2.5.1. Copying Streams

A posting sent to parent streams in a tree of streams set up by copying operations will result in copies of that posting also appearing on all the descendant streams in the tree. Such a system of streams can be built by

```
(copying parents to children for clients ...)
```

The references for the child streams are sent in an operation request posting to the parent streams where they are added to the child references of the parent. The current queue of postings held in the parent stream is copied and returned in one combined posting that is multicast to the child streams. These postings become part of each child stream. When each child receives the combined postings, it sends on to the client streams a completion posting whose value is the parent stream from which it received the posting queue. This can be used to validate that a requested copy operation has been accomplished.

2.5.2. Linking Streams

The linking operation is an optimization of copying for those cases where it is known that postings need not be retained on intermediate streams in a system of linked streams. Linking parent streams to child streams serves to restrict the parents to act only as intermediaries in a system of linked streams as in

```
(linking parents to children for clients ...)
```

The references for the child streams are multicast in an operation request posting to the parent streams. When a parent receives the references, any postings already on parent streams are sent to the children specified by the references and eliminated from the parents. Further postings are not retained on parents after they receive a linking directive but are immediately passed on to the child streams. For efficiency in forwarding, the implementation may bypass intermediate levels in a system of linked streams.

2.5.3. Value Specialization

Target specialization may also be used with the linking or copying operator to specialize the value of postings transmitted from parents to children as in

```
(linking parents to  
  (list (cons child-1 value-specialization-1) ...) ...)
```

Thereafter, all postings that traverse the links from parents to children will have the appropriate value specialization prepended to their value. This is the mechanism used to support the implicit continuations provided by the Lamina object oriented model.

2.5.4. Relocating Streams

A linking operation does not change the way that a child stream orders postings or presents them. Relocating a stream from one site to another while preserving its accumulated postings as well as its means of ordering and presenting them, is specified by

```
(relocating parents to children for clients ...)
```

This is used when there is an attempt to read from a stream that is not local to a site. The attempt causes the reference used to specify that the target stream target a new child stream, the relocation of the previously specified target. No change can be detected in the operation of eq-reference on the reference after relocation.

2.5.5. Group Streams

An application in Lamina may wish to view a group of streams as a composite, carrying out some operation only when all of the streams in the group have received a posting. To minimize unproductive scheduling, computations may wait on such composite *group streams* rather than on the individual streams. Group streams are created by *new-stream* called with a *:group* keyword argument as in

```
(new-stream tag :group member-streams)
```

A stream may be the member of only one group but a future, since its value, once attained, cannot be removed, is not so restricted. If streams of values are to be made available to several groups, a system of linked or copied streams must be used to accomplish this.

If a member stream is not local to the site of its group stream, a local member stream is created and the remote member stream is relocated there. The postings sent to the local member streams are taken from the member streams whenever a request that has been made to accept a posting from a group stream can be satisfied. Each posting available from a group stream will contain, as its value, a list of the postings received by its component streams. The order of posting elements in the list representing a group posting corresponds to the order indicated in specifying the component streams of the group stream when it was formed. Group streams are used to schedule an implicit continuation only when values are available on all streams upon which the continuation is waiting.

2.6. Creating Processes

2.6.1. Restartable Processes

A separate, concurrent computation is created by spawning the execution of a closure as in

```
(spawning #'(lambda () form) on site for clients ...)
```

The closure is formed and the clients returned immediately as the value of the spawning operation. The closure will sent to the indicated site and eventually executed there. The result of that execution will be returned to the specified clients.

Spawned computations can block waiting for a value to be available on a stream. When the value is available they will be *restarted* and any intermediate computations done previously will be redone. This approach is taken to avoid dedicating stacks for every spawned computation. However, often the continuations of partially completed computations can be spawned on the same site as their parent, thus preserving intermediate work as well as eliminating the need for dedicated stacks. This is described in Sections 3 and 5.

2.6.2. Resumable Processes

If an execution is blocked on trying to access an empty stream, it can either be restarted, as discussed earlier, or *suspended* and *resumed* when that stream receives a posting. In general, suspending and resuming a computation (without spawning continuations) requires preserving indeterminate amounts of intermediate (control and binding) state with one or more stacks. Maintaining many independent stacks is certainly an expensive operation in simulation and may also be so in a target system implementation.

However, for occasions when the full power and expense of stack switching is warranted, Lamina provides the `mounting` primitive. This is called and behaves like `spawning`, except that it creates a process with associated stack storage at the indicated site.

One could implement a multiple fork and join construct (like `cobegin` and `coend`) by mounting a number of processes with a common client stream. The creator could then wait for the appropriate number of responses on the client stream (to ensure that the other processes had completed) and then continue its execution.

In applications that wish to view executions created with `mounting` as non-terminating, the execution will typically have an initial section that sends a reference for a newly created 'task' stream to mutually agreed upon client streams (by an explicit `posting`). The referenced task stream will then be used to supply the newly mounted execution with additional operations to perform after it completes its starting procedures.

2.6.3. Remote Closures

An value may be sent to a remote site, a reference for it created there, and the reference sent to specified clients using

```
(loading value on site for clients ...)
```

The client streams are returned immediately by the form, and they will eventually receive a reference for the value loaded on the specified site.

A remote closure may be created by

```
(loading #'(lambda arglist form)
  on site for clients ...)
```

It may then be applied to locally evaluated arguments by passing it those arguments as in

```
(passing parameter-list to closure-reference
  for clients ...)
```

The result of the remote application is sent to the specified clients. The `loading` and `passing` operations are combined in `spawning`.

2.7. Miscellaneous Utilities

A few utility operations are provided by Lamina to specify computation and storage sites, dismiss computations, and provide a timeout facility for applications desiring one. Lamina also provides simulation control facilities to initiate a simulation, read the current simulation time, and do a computation without increasing the simulation time.

The function `random-site` returns a identifier for a site chosen randomly with uniform distribution over the processor sites in the simulated system. The function `random-memory` does the same thing over the memory controllers in the system. The function `local-site` returns an identifier for the site executing the function. The function `local-memory` returns an identifier for a memory controller associated with the processor on which the function is executed.

In order to provide a timeout facility, the keyword `after` followed by a number of milliseconds in simulated time may be included in functions that take Lamina keyword arguments. The simplest use might be to specify that a posting to a stream be sent at some future time.

A call to `dismiss` breaks execution. With no argument, execution is rescheduled immediately (but occurs after all previously scheduled executions are run). If an argument is specified which is a non-nil symbol, execution is terminated and will never be rescheduled. If a local stream is specified, execution is rescheduled when next that stream receives a posting—or immediately, if that stream has a posting on it.

The current simulation time in milliseconds is returned by the function `simulation-time`.

Some computations in a simulated application need not (or should not) be timed. The macro `without-clock` may be used to wrap such computations so that they are accomplished 'off the clock'. This is generally a good idea for calls to debuggers and the like as well as for input and output operations.

Something special must be done to start up a simulation. The form

```
(boot (at time site-coordinates form) (at ...) ...)
```

will spawn computations to execute forms at the indicated sites beginning at the specified times (in milliseconds). The site coordinates are given as a list, for example, '(3 2), whose length matches the represented dimensionality of the processing unit (a surface for the case shown). The `boot` construct resets the simulator and thus may only be executed as the first operation of an application being simulated. Note that `boot` spawns rather than mounts a computation. If a mounted computation is needed, it must be explicitly mounted by the computation that `boot` spawns.

3. Functional Programming

Perhaps the style of computation most readily treated as concurrent is that of functional programming. Lamina supports concurrent programming using this style by providing means to (1) spawn computations that will provide values to futures and (2) accept such values in a computation—scheduling the computation when they are available. The constructs defining the Lamina interface for functional programming are

- (`future form`) spawns execution of a *lexical closure*, that is, a procedure body to execute a given form together with an environment (determined by the rules of lexical scoping) in which to do the execution [Steele 84]. This closure is executed (eagerly) on a randomly selected site. A future which will contain the value of the computation when it is available is immediately returned.
- (`with-values future-bindings forms`) spawns an evaluation on the local site to execute the closure corresponding to the *forms*. The evaluation is done within an environment that includes bindings for given variables to the values available for the

indicated futures. The evaluation is deferred until all of the indicated futures have values that are not themselves futures. The immediate result of executing a *with-values* form is a future whose value will be supplied by the deferred evaluation.

Each element of a *future-bindings* list is itself a list: (*binding-pattern future-specifier*). If evaluation of a future specifier in a *with-values* construct produces a value other than a future, the value is encapsulated by a future. After all specified futures have values (which are not themselves futures), the values of each of the futures are destructured, that is, the values are treated as list structures and the elements of these list structures are used to bind corresponding variables in a binding pattern of arbitrary depth. These bindings will be included in the environment in which the spawned computation is executed. Only *with-values* can be used in Lamina to reduce futures to values. Values of futures are never taken as an ancillary consequence of any other operation.

The results of the evaluation spawned by *with-values* are returned as a future which will receive the value of the spawned computation. The spawned evaluation is treated as the continuation [Steele 76] of the spawning computation, and, as such, captures all stack allocated temporary variables required to execute that computation. Thus, each spawned computation may be viewed as running to completion; its continuation, if any, is an independent spawned computation.

All spawned computations run to completion (although they may be suspended by system level operations), and so the stack of the executing processor is generally left clear. Therefore any space allocated for it may be reused by the next computation on that processor, allowing the advantages of stack-based operation without incurring the space penalty discussed in Section 1. The costs of heap allocation are incurred only as needed.

4. A Simple Example: Matrix Manipulation

In this section, to illustrate the use of the Lamina functional programming interface, we will develop a parallel implementation to a simple problem in matrix manipulation. This same example problem will be used later to show the use of some of the shared variable and object oriented programming constructs. While this example is simple and by no means uses all of the Lamina programming constructs it should serve as a reasonable guide to show the form of Lamina programs and how they Lamina programs can exploit replication and pipelining to enhance parallelism.

In the ensuing program examples we have tried to use the minimum of programming constructs from the native Common Lisp language so as to focus particularly on Lamina programming. The names of Lamina programming constructs will be in upper case to distinguish them from the rest of the example code. We have also gone to some lengths to write them in a style which is probably more like the style of FORTRAN or Pascal than Lisp so as to make them more intelligible to the average reader. However, in preparing this example we have not compromised on realism. These examples do, in fact, work.

The following simple points should help the uninitiated to understand the examples.

- Lisp uses the syntax (*f x y*) to denote the application of the function *f* to the arguments *x* and *y*, i.e. *f(x, y)* in Pascal. Indeed, all syntactic forms in Lisp have the name of the programming construct as the first word inside a set of parentheses.

- The form `(defun foo (x y) «...»)` declares a function called `foo` with arguments `x` and `y` with a body `«...»`. Program constructs that define new things begin with "def". Thus, `defconstant` declares a constant.
- The form `(aref array i j)` is the array indexing operation that extracts the i^{th} row and j^{th} column value from the array, i.e. `array(i, j)` in Pascal.
- The form `(setf «place» «value»)` is an assignment statement that puts the value `«value»` into the specified place. For example, `(setf foo 42)` puts the value 42 into the variable called `foo`. Similarly, `(setf (aref array i j) 42)` puts the value 42 into the i, j^{th} element of `array`.
- Comments in Lisp programs are denoted by a semicolon character. In our example we use three to make them stand out.
- Literals are introduced in Lisp by the single quote character. For example, `'(a b c)` is the literal list with the three elements `a`, `b` and `c`.
- Functions are often passed as arguments in Lisp programs. This can be done either by passing the literal function object (code body) or by passing the name of the function. In our examples we pass functions as arguments by passing their names. A functional argument can be invoked by the use of the `funcall` function. For example, the form `(funcall «funarg» arg1 arg2...)` applies the functional argument to the arguments supplied. Thus, `(funcall 'f 42) = (f 42)`.

First let us consider the serial problem. Given a matrix `a[i, j]`, compute the new matrix `b[i, j] = f(g(a[i, j]))` where `f` and `g` are non-trivial functions and print out the resulting value. For the sake of simplicity in our example we assume the following: `f(x) = x ÷ 2` and `g(x) = x * x` and the array `a` has dimensions `width`, `height`, i.e. ranges over `a[0..width-1, 0..height-1]`. In our example, `width = 3`, `height = 3`.

Now let us make some simple global definitions that will be of use in our example.

```
(defun f (x) (+ x 2)) ;;; The function F
(defun g (x) (* x x)) ;;; The function G
(defun f-of-g-of-x (x) (f (g x))) ;;; The function f(g(x)).
(defconstant width 3) ;;; The width of the matrix
(defconstant height 3) ;;; The height of the matrix
(defconstant dimensions (list width height))
                        ;;; The dimensions of the matrix
(defun print-out-result (i j value)
  ;;; Print out a line with <i-coord> <j-coord> -> <value>
  ;;; e.g.      ;;;      1 2 -> 27
  (without-clock (format t "~&~S ~S -> ~S" i j value))
)
```

We are now equipped to define our simple, serial solution to this problem.

```

(defun compute-f-of-g-of-matrix (matrix)
  ;; This function loops over each element in the matrix, computing
  ;; f(g(x)), for each element and putting the value back in the
  ;; matrix. It ends by returning the matrix.
  (loop for i from 0 below width ;; Loop over rows
        do (loop for j from 0 below height ;; Loop over columns
                  ;; Compute new value
                  for new-value = (f-of-g-of-x (aref matrix i j))
                  ;; Update each cell in the matrix with
                  ;; f(g(<cell>)).
                  do (setf (aref matrix i j) new-value)
                )
        )
  matrix ;; Return the updated matrix
)

(defun print-out-matrix (matrix)
  ;; This function takes a matrix as its argument and loops over the
  ;; elements printing them out.
  (loop for i from 0 below width ;; Loop over rows
        do (loop for j from 0 below height ;; Loop over columns
                  ;; Print out a line with
                  ;; <i-coord> <j-coord> -> <new value>
                  do (print-out-result i j (aref matrix i j))
                )
        )
)

```

We can now execute our example program by creating the original matrix and then calling the functions defined above.

```

(defun make-example-matrix ()
  ;; Create a 2 dimensional matrix that is initialized
  ;; with the numbers 0..8
  (make-array dimensions :initial-contents
              '((0 1 2) (3 4 5) (6 7 8))
  )
)

(defun serial-example ()
  ;; Trivially compute the serial example.
  (print-out-matrix
    (compute-f-of-g-of-matrix (make-example-matrix))
  )
)

```

Now we can actually run the example.

```

(serial-example) ;; Run the example. The results follow.
0 0 -> 2
0 1 -> 3
0 2 -> 6
1 0 -> 11
1 1 -> 18
1 2 -> 27
2 0 -> 38
2 1 -> 51
2 2 -> 66

```


4.1. Using Lamina's Functional Programming Primitives in our Matrix Example

In this implementation of our example problem we use the CARE functional programming constructs `FUTURE` and `WITH-VALUES` to compute our solution in parallel. For each cell in the matrix we spawn a future to compute the desired value. This gives us parallelism through replication.

```
(defun compute-f-of-g-of-matrix-as-futures (matrix)
  (loop for i from 0 below width ;; Loop over rows
        do (loop for j from 0 below height ;; Loop over columns
                  ;; Compute new value as a future
                  for new-value
                    = (FUTURE (f-of-g-of-x (aref matrix i j)))
                  ;; Update each cell in the matrix with
                  ;; f(g(<cell>)).
                  do (setf (aref matrix i j) new-value)
                )
        )
  matrix ;; Return the updated matrix
)
```

We now have to use the CARE simulator to run this example. The CARE simulator like many operating systems cannot simply execute the program without having a little more information about the way in which it is to be launched. We therefore have to write a little harness to execute our example so that it will be launched on a particular processor at a certain time. A similar piece will be used to boot each one of our examples below.

```
(defun boot-functional-programming-example-1 ()
  ;; Starts up the example on processing element (1 1) at time = 0.
  (BOOT (at 0 '(1 1)
           ;; The following actually executes the example.
           (print-out-matrix
            (compute-f-of-g-of-matrix-as-futures
             (make-example-matrix)
            )
          )
  )
)

(boot-functional-programming-example-1) ;; Run the example
0 0 -> #[remote (1 1) #[future 201333 (f (g (aref matrix i j))) 0 0]]
0 1 -> #[remote (1 1) #[future 270998 (f (g (aref matrix i j))) 0 0]]
0 2 -> #[remote (1 1) #[future 299662 (f (g (aref matrix i j))) 0 0]]
1 0 -> #[remote (1 1) #[future 332660 (f (g (aref matrix i j))) 0 0]]
1 1 -> #[remote (1 1) #[future 366658 (f (g (aref matrix i j))) 0 0]]
1 2 -> #[remote (1 1) #[future 400989 (f (g (aref matrix i j))) 0 0]]
2 0 -> #[remote (1 1) #[future 437655 (f (g (aref matrix i j))) 0 0]]
2 1 -> #[remote (1 1) #[future 470986 (f (g (aref matrix i j))) 0 0]]
2 2 -> #[remote (1 1) #[future 500983 (f (g (aref matrix i j))) 0 0]]
```

Note: in this case what gets printed out is a collection of futures. This is because the Lamina system, unlike most parallel systems with futures, does not perform automatic defuturing of any futures.

If we want the actual values associated with the futures then we must wait for the computation to take place. To do this we use the `WITH-VALUES` construct. We could modify our printing procedure as follows to wait for the results.

```

(defun print-out-matrix-of-futures (matrix)
  (loop for i from 0 below width ;; Loop over rows
    do (loop for j from 0 below height ;; Loop over columns
      do (WITH-VALUES ((value-of-future-in-cell
                        (aref matrix i j)
                        )
                        )
        ;; Update the matrix with the defutured
        ;; value.
        (setf (aref matrix i j)
              value-of-future-in-cell
              )
        ;; Print out a line with
        ;; <i-coord> <j-coord> -> <new value>
        (print-out-result i j (aref matrix i j))
        )
      )
    )
  )

(defun boot-functional-programming-example-2 ()
  ;; Starts up the example on processing element (1 1) at time = 0.
  (BOOT (at 0 '(1 1)
    ;; The following actually executes the example.
    (print-out-matrix-of-futures
      (compute-f-of-g-of-matrix-as-futures
        (make-example-matrix)
      )
    )
  )
)

```

Now we can run the example.

```

(boot-functional-programming-example-2)
0 0 -> 2
0 1 -> 3
0 2 -> 6
1 0 -> 11
1 1 -> 18
1 2 -> 27
2 0 -> 38
2 1 -> 51
2 2 -> 66

```

Note: in this case, whilst we are printing out the values in the matrix, other values are being evaluated. This is a form of pipeline execution. The performance of this example is, however, limited by the fact that a single process is waiting for all of the results before it can print them out. A more parallel solution would result if we could avoid this synchronization to print out the answers.

The code for this example is written using single-valued futures. A consequence of this future-based functional programming approach is that many short-lived dynamic references are created and then abandoned when the result appears. Reclaiming the space allocated for them requires the expensive global synchronization discussed in Section 1. One way to improve things might be to establish a stream to the process that prints out the answers so that we could print them out as they arrive with the minimum of delay. This would mean

that only the reference to this reply stream would have to be created, not one reference to each of the futures that represent the matrix. To do something like this, though, we would have to modify our simple use of single-valued futures in our example. This could be done either by introducing a new type of future that represents a sequence of values or we could treat these sequences of values as communication channels between objects. The idea of streams as sequences of values and tasks to perform is developed further in the next section in which we discuss an object-oriented implementation of our matrix processing example. The idea of a Multi-future, a multi-values future that receives unordered values from a number of processes, is used by the Poligon system to support its Bag data type.

5. Object Oriented Message Passing

The Lamina object programming model is founded on the notion of asynchronously communicating objects. An object, as used here, is a collection of variables—its *state variables*—manipulated by (and only by) a set of procedures—the *methods* associated with that object. Objects may be defined within a compiled class inheritance network; the current implementation uses the inheritance facilities of *Flavors* [Weinreb 81].

A Lamina object is allocated in *local* space and is referenced externally by its *task stream*, a stream maintained as one of its state variables. This task stream is referred to as the object's *self-stream*. It is typically the reference to this stream that is used to represent an object remotely. The information placed on this stream (that is, provided as its values) specifies *tasks* for the object; each unit of information is called a *message*. A message is internally structured as a *task request posting*, whose value consists of a *task selector* symbol that identifies the method to execute, along with the associated parametric values for the execution.

5.1. Computational Flow

As illustrated in Figure 2, the messages arriving on an object's task stream specify tasks to be performed by that object. Every object has a spawned *dispatch process* associated with it that removes and executes each message on its task stream in turn. Tasks usually mutate the state variables of the object and generate new messages. They have exclusive access to their environment (i.e., state and temporary variables) during execution.

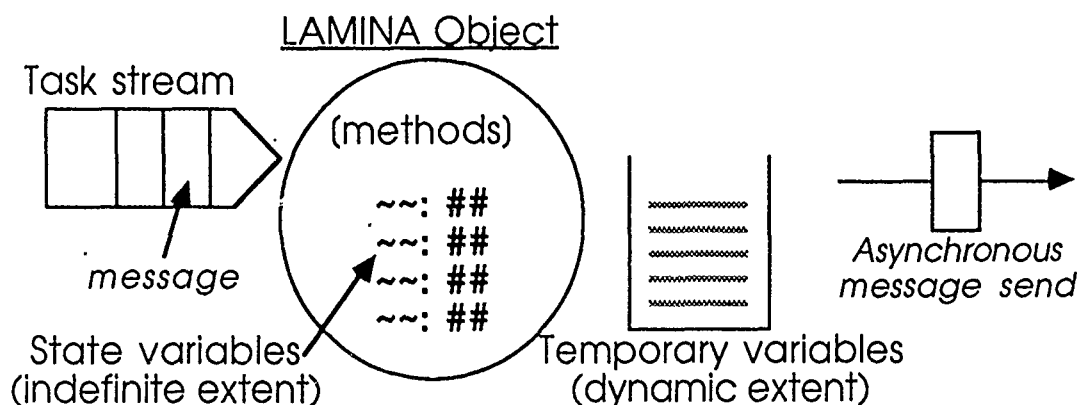


Figure 2. Message passing model

Tasks are *data driven* in that they are started only when all the needed information is available. Typically, a single message, in conjunction with the object's state variables, contains all the relevant information for the task execution. Tasks are generally intended to be accomplished as the stages of pipelines that organize the work performed by the objects

of the application. In order not to block the pipeline, a task, once started, is run to completion.

5.2. Providing Atomicity

Although Lamina provides the programmer with a run-to-completion model, there may be system reasons for preempting a task, for example, to handle a debug trap or because the task's run quantum has expired. When this occurs, the object does not execute any other tasks until the preemption is resolved. This prevents other tasks on that object from gaining access to the environment of the suspended task. However, since other objects may execute tasks during this time, true atomicity can only be enforced if no state is shared between execution environments. The mechanism by which objects communicate ensures this.

Lamina objects can never share state because they only communicate by exchanging messages containing *independent* copies of local structures. Furthermore, the state variables of an object are only visible to its own methods and are therefore only accessible within a private task. Thus the atomicity of operations on an object is preserved even in the presence of preemptions.

5.3. Sending a Task Request

Sending a task request message in Lamina is non-blocking so as to accommodate pipelined operations on objects directly. The construct for asynchronously sending a message is `send`, which takes as arguments one or more target task streams, a task selector symbol, and a list representing the parameters to be provided to the task executions. Since `send` is no more than syntactic sugar for the `post` primitive, the sender may provide additional control or debugging information as described in Section 2.

The value immediately returned by `send` is the list of clients supplied. As a convention, the *clients* may expect to receive consequent task requests later in the computation.

5.4. Defining Objects

Lamina object types are built upon the base `Flavor, Lamina`, which defines the instance variable, `Self-Stream` that stores its task stream. The default kind of task stream is a normal unordered stream; the `'mixin'` flavors `ordered-self-stream` and `sequenced-self-stream`, are provided to override this default.

5.5. Trigger Methods

The 'top level' methods executed as tasks by Lamina objects are called *triggers*. They are defined using the `deftrigger` form as shown below. The parameter list provided to `deftrigger` corresponds to the value (and the other information, which can be optionally ignored) contained in each `posting` received on its task (self) stream. In particular, the parameter specification may be used to destructure the value provided, as is done in the example.

5.6. Creating Objects

The form

```
(creating type initializations for clients on site ...)
```

stipulates the creation of a object on the indicated site, or on a randomly selected site if none is indicated. When the creation has been accomplished, the client streams will receive a posting whose value is the task stream of the created object.

The *initializations* are a list of alternating keywords (corresponding to the state variable names for the object being created) with their initial values. These values are computed in the context of the object requesting creation. As an example, a creating form is included in the `create-an-object` function defined below. The function `create-self-stream` is provided to create a stream as defined by the type of the Lamina object being created. This can be used, as in the example, to create the object's task stream before the actual object has been created. If the initializations contain the specification `:Self-Stream` then the object uses a stream that has been created for it externally by `create-self-stream` as its self-stream.

5.7. An Object-Oriented Matrix Manipulation Example

The same matrix manipulation problem as that used above can be recoded in an object-oriented fashion. We could have a matrix of objects representing the values in each cell of the matrix and leave it to the objects themselves to solve the sub-components of the problem, i.e. apply functions such as `F` and `G` to the value in the cell.

In this case we declare a class of objects, which we call `Function-Executer`. Each cell of the matrix will be represented by an instance of this class. Instances of the class `Function-Executer` know about both their location within the matrix and the value in the cell of the matrix. We will define behavior for this class so that there will be two methods that are used. The first, called `:Update-Value-By-Evaluating-Function`, takes a function its argument and applies it to the value in the cell of the matrix, transforming the value, for example, from $x \rightarrow f(x)$. The second method defined is `:Execute-Procedure`, which is much like the `:Update-Value-By-Evaluating-Function` except that rather than updating the value in the cell it merely calls its argument function for any side-effects it might perform. In our case we use this to print out the result.

First we define the class of objects to use, the means of creating them individually and then a procedure to create a matrix of them.

```
;;; Declare the class of object that represents a value and knows
;;; how to execute functions using that value as an argument.
(defflavor function-executer
  ;;; Instance variables.
  (x-coord ;;; The row in the matrix
   y-coord ;;; The column in the matrix
   value) ;;; The value in this cell of the matrix
  (lamina) ;;; Based on the class called "Lamina", a
             ;;; primitive concurrent object class.
  :initable-instance-variables ;;; Used for initialization only.
)
```

```

(defun create-an-object (flavor x y value)
  ;;; Creates an object of a specified flavor (class) to represent
  ;;; the cell x, y in the matrix, at which we find the Value.
  (let (stream) ;;; Declare a local variable called Stream
    (setf stream (CREATE-SELF-STREAM flavor))
    ;;; We explicitly create the Self-Stream for this object so
    ;;; that we can record it in our matrix of objects.
    ;;; Creates an object of the specified flavor on a random
    ;;; site with the specified self-stream, initializing the
    ;;; values of the coord and value instance variables.
    (CREATING flavor
      ;;; A list of initialization arguments for the creation
      ;;; of the instance
      `(:self-stream ,stream :value ,value :x-coord ,x
        :y-coord ,y
      )
    )
    ;;; Return the stream. This is our pointer to the remote
    ;;; object.
    stream
  )
)

(defun make-example-matrix-of-objects (flavor)
  ;;; Create a 2 dimensional matrix that is initialized with
  ;;; function executing objects that represent the numbers 0..8.
  (let ((matrix (make-array dimensions)))
    (loop for i from 0 below width
      do (loop for j from 0 below height
        ;;; Create an object and put it into the
        ;;; specified cell in the matrix.
        do (setf (aref matrix i j)
          (create-an-object flavor i j
            (+ (* i 3) j)
          )
        )
      )
    )
    ;;; Return the matrix.
    matrix
  )
)

```

Now that we can create our matrix of objects — actually a matrix of remote-addresses to the self-streams of objects we can define the behavior of the objects by declaring the methods mentioned above.

```

(DEFTRIGGER
  (function-executer :update-value-by-evaluating-function)
    ((function) clients &rest ignore)
  ;; Method that tells function-executer objects how to perform the
  ;; operation
  ;; value := f(value),
  ;; where f is the argument Function and Value is the
  ;; instance variable denoting the value associated with the
  ;; object. The Clients argument is ignored because we execute
  ;; this method simply for effect, we do not send any value on to
  ;; anyone else.
  (ignore clients)
  (setf value (funcall function value))
)

(DEFTRIGGER (function-executer :execute-procedure)
  ((function) clients &rest ignore)
  ;; Method that tells function-executer objects how to perform the
  ;; operation f(x-coord, y-coord, value), discarding the result of
  ;; the operation, where f is the argument Function and x-coord,
  ;; y-coord and Value are the instance variables denoting the
  ;; position of the cell in the matrix and the value associated
  ;; with the object. The Clients argument is ignored because we
  ;; execute this method simply for effect, we do not
  ;; send any value on to anyone else.
  (ignore clients)
  (funcall function x-coord y-coord value)
)

```

Now we can define the procedure that tells the individual elements of the matrix to update themselves by applying the function f-of-g-of-x.

```

(defun update-matrix-of-objects-f-of-g-of-x (matrix)
  ;; Tells each object in the matrix of objects to change the value
  ;; that it represents (x) to f(g(x)).
  (loop for i from 0 below width ;; Loop over rows
    do (loop for j from 0 below height ;; Loop over columns
      ;; Send a message to each cell invoking the
      ;; :Update-Value-By-Evaluating-Function method.
      do (SENDING (aref matrix i j)
        :update-value-by-evaluating-function
        ;; F-Of-G-Of-X is the function to be
        ;; invoked by each object.
        '(f-of-g-of-x)
      )
    )
  )
  matrix ;; Return the updated matrix
)

```

Now we must specify how to print out the results. This is done in much the same way as the updating process above.

```

(defun print-out-matrix-of-objects (matrix)
  ;; Tells each object in the matrix of objects to execute the
  ;; function Print-Out-Result on the value it represents.
  (loop for i from 0 below width ;; Loop over rows
        do (loop for j from 0 below height ;; Loop over columns
                  do (SENDING (aref matrix i j) :execute-procedure
                              '(print-out-result)
                              )
                )
        )
  )
)

```

Now we declare the bootstrapping harness for the example and execute it.

```

(defun boot-object-oriented-example-1 ()
  ;; Starts up the example on processing element (1 1) at time = 0.
  (BOOT (at 0 '(1 1)
            ;; The following actually executes the example.
            (print-out-matrix-of-objects
              (update-matrix-of-objects-f-of-g-of-x
                (make-example-matrix-of-objects
                  'function-executer
                )
              )
            )
        )
  )
)

(boot-object-oriented-example-1) ;; Run the example
0 0 -> 2
0 1 -> 3
0 2 -> 6
1 2 -> 27
2 1 -> 51
2 2 -> 66
1 1 -> 18
1 0 -> 11
2 0 -> 38

```

Note: in this case we have the results appearing non-deterministically. Strictly speaking this example will not work properly in the general case, since we cannot guarantee that the :Update-Value-By-Evaluating-Function message is received and processed before the :Execute-Procedure message for any given cell. We should therefore either recode this problem so that the combined update and print instruction is sent all at once, or we could use strictly ordered streams for the messages sent to the cells of the matrix. On the other hand, we could use the pipelined method shown below.

5.8. A Pipelined Object-Oriented Example.

We could, in contrast to the object oriented implementation above, recast the problem so as to pipeline the operations to any depth as is shown in this example, which is structurally much like the last one (See Figure 3). The primary difference is that, instead of each cell of the matrix in question pointing to a single object that knows how to execute functions, each cell in the matrix now points to an object which is now at the front of a chain of objects each of which knows how to execute functions. This allows each element in the chain (stage in the pipe) to perform a small amount of work and then pass its results, along with

the instructions about the rest of the work to be performed, to the succeeding elements in the chain.

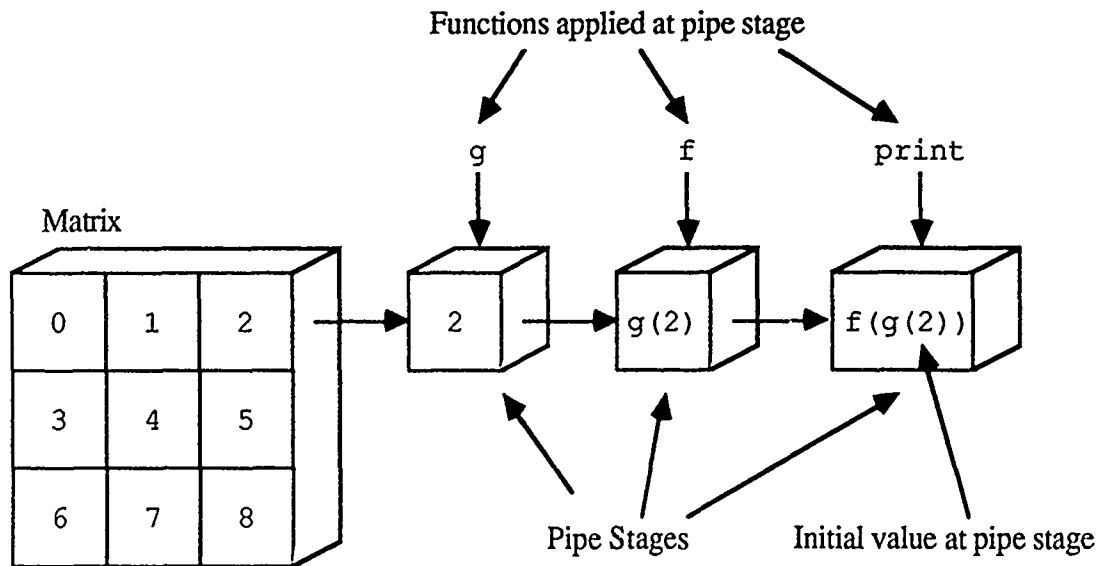


Figure 3. A Pipelined, Object-Oriented Implementation. Each element of the matrix points to a pipe of function executing objects (element [0, 2] shown here). Values pass down the pipe as they are computed and successive functions are applied to them

In this example we define a new class (flavor) of objects to do the processing, which we call Pipe-Element. The method :Execute-Pipe-Stage is used to accept a list of functions as its argument and a value. Any one stage in the pipe applies one function to the incoming value and then passes the resulting value and the remaining functions on to the next stage in the pipe.

```

;;; Declare the class of object just like Function-Executer only
;;; it also has a pointer to the next element in the pipe.
(defflavor pipe-element
  ;;; Instance variables.
  ((next-in-pipe nil)) ;;; The next stage in the pipe.
                        ;;; This initialized to the null
                        ;;; value. If we need another
                        ;;; stage in the pipe we will fill
                        ;;; in this slot.
  (function-executer) ;;; A subclass of Function-Executer
  :initable-instance-variables ;;; Used for initialization.
)

```

```

(DEFTRIGGER (pipe-element :execute-pipe-stage)
  ((incoming-value function-to-apply
    &rest further-functions
    )
   clients &rest ignore
  )
  )
  ;;; Defines a method called :Execute-Pipe-Stage, which is rather
  ;;; like the methods defined above. It updates the Value instance
  ;;; variable by applying a function passed to it as an argument to
  ;;; the coordinates and the value. If the value has not been set
  ;;; yet then this is passed in as an argument also. The arguments
  ;;; to the method also specifies a list of further functions to
  ;;; execute. If this is specified then there are further stages
  ;;; in the pipe to execute. If we have not already created an
  ;;; object for the next stage in the pipe then we do so. We
  ;;; then send the next stage in the pipe the remainder of the work
  ;;; to be performed. In this example, the argument
  ;;; "(incoming-value function-to-apply &rest further-functions)"
  ;;; specifies that the arguments are represented as a list, whose
  ;;; first element is the incoming value with which we are to
  ;;; compute, its second value is the function to execute and any
  ;;; remaining values are kept in a list called further-functions.
  ;;; As we invoke this method on successive stages in the pipe, the
  ;;; function to apply is not passed along to the next stage
  ;;; so that, for stage[n] in the pipe, the function-to-apply is
  ;;; the first element in the list of further-functions for
  ;;; stage[n-1]. When the list of further-functions becomes empty
  ;;; we stop.
  (ignore clients)
  (setf value (funcall function-to-apply x-coord y-coord
    (or incoming-value value)
    )
  )
  )
  ;;; When there are elements in the list of further-functions
  ;;; then continue down the pipe.
  (when further-functions
    ;;; Create the next stage in the pipe if we have
    ;;; not done so yet.
    (when (not next-in-pipe)
      (setf next-in-pipe
        (create-an-object 'pipe-element x-coord y-coord nil)
      )
    )
    ;;; Send the remaining functions still to be executed
    ;;; to the next stage in the pipe to deal with.
    (SENDING next-in-pipe :execute-pipe-stage
      (cons value further-functions)
    )
  )
  )
  )
  )

```

Now we define two new functions F' (f-primed) and G' (g-primed) which are just like F and G except they take the x and y coordinates of our cell in the matrix as arguments as well. Doing this allows us to have all of the functional arguments that are passed to the different stages in the pipe have the same number of arguments. Enforcing this symmetry obviates any complications due to making sure that we are calling the functions with the right number of arguments.

```

(defun f-primed (x y value)
  ;; Just like the function F, only it is also passed the
  ;; coordinate arguments which it ignores.
  (ignore x y)
  (f value)
)

(defun g-primed (x y value)
  ;; Just like the function G, only it is also passed the
  ;; coordinate arguments which it ignores.
  (ignore x y)
  (g value)
)

```

Now we can define the actual functions that get things going; the top level loop that starts up the different pipe stages and the bootstrapping harness.

```

(defun pipe-example (matrix)
  ;; Is passed a matrix of objects each of which represents the
  ;; head of a pipe. It starts execution by telling each pipe to
  ;; execute the G-Primed, F-Primed and then the Print-Out-Result
  ;; functions in successive stages of the pipe.
  (loop for i from 0 below width ;; Loop over rows
    do (loop for j from 0 below height ;; Loop over columns
      ;; Send an :Execute-Pipe-Stage message to each
      ;; cell in the matrix
      do (SENDING (aref matrix i j) :execute-pipe-stage
        '(nil g-primed f-primed
          print-out-result
        )
      )
    )
  )
  matrix ;; Return the updated matrix
)

(defun boot-pipelined-object-oriented-example-1 ()
  ;; Start up a pipe-lined version of the object matrix example.
  (BOOT (at 0 '(1 1)
    ;; The following actually executes the example.
    (pipe-example
      (make-example-matrix-of-objects 'pipe-element)
    )
  )
)

```

Now we can actually run the program.

```

(boot-pipelined-object-oriented-example-1) ;; Run the example.
0 0 -> 2
0 1 -> 3
0 2 -> 6
1 2 -> 27
2 1 -> 51
2 2 -> 66
1 1 -> 18
1 0 -> 11
2 0 -> 38

```

In fact, if we add a loop around the initial loop, we can continually pump values into this nine-wide (3 by 3), three-stage pipe. If we could achieve perfect balance in our pipe we could hope for a speedup of the order of width * height * <number-of-stages-in-pipe>, which in this case = 3 * 3 * 3. We can do this with this simple rewrite to the pipe-example function.

```
(defun pipe-example (matrix)
  (loop do
    (loop for i from 0 below width ;;; Loop over rows
      do (loop for j from 0 below height
        ;;; Loop over columns
        do (SENDING (aref matrix i j)
          :execute-pipe-stage
          '(nil g-primed f-primed
            print-out-result
          )
        )
      )
    )
    (dismiss) ;;; Allow other processes on this site to have
    ;;; a look in.
  )
)
```

Similarly, because the same message is being sent to each stage in the pipe we could, in fact do this using multicast communication. This involves making a list of all of the target objects and then sending the message. We could therefore rewrite the pipe-example function above as follows:

```
(defun pipe-example (matrix)
  (loop for targets =
    (loop for i from 0 below width
      ;;; Loop over rows gluing the lists together.
      append (loop for j from 0 below height
        ;;; Loop over columns, collecting
        ;;; up the elements.
        collect (aref matrix i j)
      )
    )
    do (SENDING targets :execute-pipe-stage
      '(nil g-primed f-primed print-out-result)
    )
    (dismiss) ;;; Allow other processes on this site to have
    ;;; a look in.
  )
)
```

5.9. An Object-Oriented, Pipelined example with Final Synchronization.

Now, let us consider the problem of wanting to consolidate all of the results from our computations. It may not be of any use to us to create a set of pipes with useful values at their ends. We need to return a matrix of values, and perhaps print it out. For this we create a new object, which we call the collector, which both starts the process and is the eventual target for the results that are computed.

To do this we create a new flavor called collector of which we only create one instance, the collector itself. The collector knows how to handle two types of message; :start and

returned-result. The :Start method is invoked when the program is booted, is fires up each of the pipes for the matrix. The returned-result is invoked by each pipe when it has got to the end of its processing. The returned-result notifies the collector object of the result and, when enough results have been accumulated, the answers are printed out.

```

;;; Defines a class called Collector of which we will only
;;; make one instance.
(defclass collector
  (instance-variables
    (matrix      ;; The matrix of pipes
      answers-outstanding
        ;; The number of replies we are still expecting.
      answers    ;; A matrix of answers.
    )
    (function-executer)
    ;; Based on the class Function-Executer.
  )

  (DEFTRIGGER (collector :start) (arguments clients &rest ignore)
    ;; Defines a method called :Start, which starts off the process,
    ;; creating the matrix of pipes and a matrix in which to store
    ;; the results as they come in. The instance variable
    ;; Answers-Outstanding is initialized to the number of answers
    ;; that we are expecting. It will be decremented as each
    ;; result comes in and when this hits zero we will print out
    ;; the resulting table. The pipes are started off executing
    ;; the method :Execute-Pipe-Stage-Finally-Returning-Answer
    ;; (defined below), which is a modified version of the
    ;; :Execute-Pipe-Stage method defined above. Note: the
    ;; argument ":For self-stream" means that we are passing the
    ;; self-stream of the collector object as an explicit client
    ;; to each of the pipes. This is the stream that will be
    ;; used to notify the collector of the results as they are
    ;; computed.
    (ignore clients arguments)
    (setf matrix (make-example-matrix-of-objects 'pipe-element))
    (setf answers (make-array '(3 3)))
    (setf answers-outstanding 9)
    (loop for i from 0 below width ;; Loop over rows
      do (loop for j from 0 below height ;; Loop over columns
        do (SENDING (aref matrix i j)
          :execute-pipe-stage-finally-returning-answer
            '(nil g-primed f-primed)
            :for self-stream
          )
        )
      )
    )
  )
)

```

```

(DEFTRIGGER (collector :returned-result)
  ((i j result) clients &rest ignore)
  ;;; Defines a method called :Returned-Result. This is invoked
  ;;; when the pipe has finished executing its work. It stores
  ;;; the result away in the appropriate slot in the answers
  ;;; matrix and decrements the answers-outstanding counter
  ;;; so that when we hit zero we will print out the results.
  (ignore clients)
  (setf (aref answers i j) result)
  (setf answers-outstanding (- answers-outstanding 1))
  (when (= answers-outstanding 0)
    (print-out-matrix answers)
  )
)

```

Now that we have fully specified the behavior of the collector we can add the extra necessary behavior to the pipe elements. In this case we simply add a new method called :execute-pipe-stage-finally-returning-answer, which is pretty much like the :execute-pipe-stage method declared above only when we get to the end of the pipe and there are no more functions to execute we send a :returned-result message to the collector object which is passed down the pipe as the client of the computation.

```

(DEFTRIGGER
  (pipe-element :execute-pipe-stage-finally-returning-answer)
    ((incoming-value function-to-apply
      &rest further-functions)
     clients &rest ignore)
    ;;; Defines a method called
    ;;; :Execute-Pipe-Stage-Finally-Returning-Answer which
    ;;; acts just like :Execute-Pipe-Stage, only when it runs out
    ;;; of things to do at the end of the pipe it returns the
    ;;; resulting value to the collector object, which is passed
    ;;; in as a client. Computes the new value for the Value
    ;;; instance variable by applying the functional argument.
    (setf value (funcall function-to-apply x-coord y-coord
      (or incoming-value value)
    )
    )
  )
  (if (not further-functions)
    ;;; Then we have got to the end of the pipe.
    ;;; Return the result to the collector object (clients).
    (SENDING clients :returned-result
      (list x-coord y-coord value)
    )
    ;;; Else maybe create the next stage in the pipe and
    ;;; tell if what to do.
    (when (not next-in-pipe)
      (setf next-in-pipe
        (create-an-object 'pipe-element x-coord y-coord nil)
      )
    )
    (SENDING next-in-pipe
      :execute-pipe-stage-finally-returning-answer
      (cons value further-functions) :for clients
    )
  )
)

```

```
)
)
```

Now all that we need do is declare the top level functions that start things off. These consist of a simple function that creates the collector object and sends it a :start message and the bootstrapping harness.

```
(defun create-and-start-collector ()
  ;; Creates the initial collector object and sends it a
  ;; :start message to get the ball rolling.
  (let ((collector-stream (CREATE-SELF-STREAM 'collector)))
    (CREATING 'collector `(:self-stream ,collector-stream))
    (SENDING collector-stream :start nil)
  )
)

(defun boot-pipelined-object-oriented-example-2 ()
  ;; Start up a pipe-lined version of the object matrix
  ;; example with synchronization at the end.
  (BOOT (at 0 '(1 1)
    ;; The following actually executes the example.
    (create-and-start-collector)
  )
)

(boot-pipelined-object-oriented-example-2) ;; Run the example
```

5.10. Implicit Continuations

For Lamina objects, continuations of a computation are often some explicit trigger method of some explicit object. There are cases, however, in which it is inconvenient to create an explicit name for a continuation. As a syntactic construct, execution of a continuation of a computation can be specified to occur in the context of an executing object (as defined by its set of state variables and the environment of the continuation) each time that postings have been received on some given streams. The execution spawning the continuation is finished normally and then the next operation to be done on the object is taken from its task stream without delay. Thus Lamina objects can be viewed as *monitors* [Andrews 83] (because the independently atomic operations on objects give the required mutual exclusion) but operations on them are unnested. This is done to facilitate pipelined operation: task request postings queued for operation on an object are not deferred for a pending continuation.

The construct

```
(with-postings stream-bindings form)
```

creates an implicit continuation in the context of an object. The *stream-bindings* is a list, each element of which is a list of a *binding-pattern* and a *stream*. Each of the postings on the indicated streams (including the posting clients, tag, key, origin, and properties) will be destructured and bound to a corresponding variable identifier according to the associated *binding-pattern*. These variables and associated values are also part of the execution environment of the continuation.

As an example of the use of with-postings, consider the example shown below. It uses nested with-postings constructs to create continuation closures that first create and collect pairs of lamina objects and then distribute requests on an input stream to the collected

triples in a round robin fashion. Note that instance variables may be accessed by the continuations.

```
(DEFTRIGGER (distributor :start-servers) ((count input-stream))
;;; Round robin distribution of input requests to created
;;; server pairs"
  (let ((servers nil)
        (as (CREATING 'a '() for (NEW-STREAM)
                        on (RANDOM-SITES count)))
        (bs (CREATING 'b '() for (NEW-STREAM)
                        on (RANDOM-SITES count))))
    (WITH-POSTINGS ((a as) (b bs))
      (cond
        ((null servers) ; first invocation of continuation
         (setf servers (circular-list (list a b))) ; single elt
         (WITH-POSTINGS ((request input-stream)
                         ;;; start distributor
                         (SENDING (pop servers) :request request))) ; multicast
         (:else ; other invocations, upto count
          (push (list a b) (cdr servers)))))))
```

Figure 4. Continuation Closures. [1] References for streams on which responses are expected are sent in task request postings to other objects as places to supply response postings. [2] Intermediate variables (that is, the environment) and a pointer to a block of code required to execute the form wrapped in a with-postings construct are captured in a continuation closure, attached to a stream, and linked to the stream(s) on which responses are expected. [3] When all required postings become available on these streams, [4] the response postings together with the closure are sent to the task stream of the object that generated the closure. The closure is executed (in its turn) atomically within the context of the object and lexical environment of the form. Variable bindings are made as specified to the elements of the available response postings. Note that the execution that spawned execution of the closure and the execution so spawned are independently atomic. The state variables of the object and any structures they reference can be changed by some other operation taken from the task stream between the two executions. The syntactic convenience is only that: invariants that need to be preserved across independent executions need to be met at the boundaries between the execution that spawned execution of the closure and the execution so spawned.

The implicit continuation will be executed atomically with respect to any other operations on the indicated object and in the context of its state variables and the lexical environment in which the form appears. A schematic of the mechanism supporting implicit continuations in objects is shown in Figure 4.

5.11. Lamina Objects and Actors

The Lamina object model is similar to Actors [Hewitt 73], in that message arrival triggers computation and message arrival order is non-deterministic. However, it departs from Actors in a number of ways, primarily by trading off flexibility for efficiency.

- Not everything is an object. Predefined data types such as numbers, symbols, arrays and cons cells exist as primitives, and operations on them do not entail message-passing. Although structures are passed by copying, they are locally mutable.

- Streams are first-class entities independent of objects. Objects may establish communications over streams other than their task streams. Streams may also be shared between objects as described in Section 2.
- The default operation of a Lamina object is serial command execution. For serial execution sequences, stack allocation of dynamically allocated structures can be used where the compiler can determine that references to the structures have dynamic extent.
- Mutation is explicit. Unlike actors, Lamina objects do not deal with state changes by specifying a *replacement* [Hewitt 73] actor for themselves, but rather explicitly manipulate their own state variables through assignment.
- Although structures are passed by copying, they are locally mutable. Tasks may change them and pass the changed structure to some other object. The copying done to transmit the structure will occur asynchronously with method execution.
- Finally, Lamina relies on compiled inheritance for method combination rather than upon runtime delegation, and it does instantiation by compiled template rather than by copying a selected instance with specified exceptions.

6. Shared Variables

Shared variables in Lamina are cells that are managed by a memory controller and whose associated value may be mutated. Lamina also supports shared data pairs ('conses') and arrays. A shared variable reference is constructed, accessed, and mutated by the interface operations described in this section. For all these operations, execution is deferred and no other executions are performed by the initiating processor until the indicated operation is accomplished.¹

Shared queues (which are modelled using streams) are also provided. These queues are maintained in a processor's local memory. When a process reads from a shared queue, it is halted and descheduled; execution is resumed when the requested data arrives.

6.1. Creating and Accessing Shared Variables

A single shared variable can be allocated and initialized using the `shared-variable` operator, that takes as a required argument the initial value for the shared variable, and creates and returns a reference to a cell containing the indicated value. The value of the cell, in general, must be a self-referential datum or a dynamic or static reference.

An optional argument can be used to specify a memory site at which to allocate the cell; if it is omitted, a randomly selected memory site is chosen. Alternatively, the macro `(in-memory site-identifier ...)` can be used to specify a default site for all allocations (for simple as well as structured shared variables) performed within its dynamic scope.

Once a shared variable has been allocated, the following constructs may be used to access or alter its value:

¹Note that, because the simulator is executing in a uniprocessor environment, a stack must be maintained for each deferred execution. Thus executions must be resumable (not merely restartable) to use the shared variable Lamina interface. This is discussed in Section 2.

- `(shared-read cell)` retrieves the value of the referenced cell.
- `(shared-write cell value)` modifies the value of the referenced cell. The new value is returned.
- `(shared-exchange cell value)` performs the same function as `shared-write`, except that the prior value of the reference is returned. This exchange is atomic.
- `(shared-replace-conditional cell old new)` atomically compares the contents of the referenced cell with *old*, and, if they are identical, replaces the contents with *new*.

For each of these constructs, the operation is guaranteed to be completed before execution is resumed.

6.2. Shared Data Structures

Lamina also provides support for shared data structures, namely shared pairs and shared arrays. Shared pairs form the foundation of linked data structures such as lists and graphs.

The constructor

```
(shared-cons car-value cdr-value)
```

creates and initializes a shared pair, returning a reference to it. The accessors are, naturally, `shared-car` and `shared-cdr`, while the mutators are `shared-rplaca` and `shared-rplacd`. Also, the form

```
(cache-shared-pair shared-pair-reference)
```

may be used to make a local, that is, non-shared, copy of a shared pair in local space.

The form

```
(shared-array dimension...)
```

returns a reference to a shared array. The *dimensions* argument is a list of positive integers, denoting the size of each dimension of the array. There are optional `:initial-element` and `:initial-contents` keyword arguments, which may be used, respectively, to initialize all the elements of the array to the single value specified or to initialize each element of the array to the value of the corresponding element in a list or a list of lists. Shared arrays are initialized to `nil` by default.

The accessor `shared-aref` reads elements of the shared array. The mutator `shared-aset` writes array elements. Both operations are bounds-checked against the dimensions of the array. Finally, the `cache-shared-array` function returns a local (non-shared) copy of the referenced shared array, while `fill-shared-array` copies data from a local array into a shared array.

6.3. Shared Queues

A shared queue construct, which is implemented as a Lamina stream, is also provided. Shared queues are managed by a processor which provides atomic access to the queue and, when the queue is empty, maintains a FIFO queue of processes requesting data from it; the requests are serviced when data is added to the queue. Further, whenever a process

attempts to remove data from the queue, the process is descheduled; execution is rescheduled when the requested data arrives.

Shared queues are created by the `shared-queue` function, which takes one optional argument representing the queue's tag, which may be used for debugging. Items may be added to the queue with the `shared-enqueue` function. The `shared-dequeue` function removes and returns the top item of the queue, while the `shared-queue-top` function merely returns it². A `shared-queue-p` predicate is also provided to test whether an item is a shared queue.

Unlike other shared variable operations, accesses to shared queues do not cause the initiating processor to stall waiting for completion. A process executing `shared-enqueue` continues immediately, without waiting for the data to arrive on the queue. A process which accesses a queue, using `shared-dequeue` or `shared-queue-top`, will be halted and descheduled. Execution is rescheduled when the data arrives, but the initiating processor may perform other executions in the meantime.

6.4. Other Synchronization

A simple spin lock is provided for busy-wait synchronization. A lock is implemented as a cell that is initialized to a value other than `nil`, and the atomic exchange operation is used to set and clear it. The form

```
(with-spin-lock lock form)
```

executes the given form after acquiring the referenced lock; subsequently, the lock is released and the value produced by the execution of the form is returned.

Such a synchronization operator might be used in incrementing a shared counter as in³

```
(defun locked-increment (<counter> <lock> &optional (delta 1))
  (WITH-SPIN-LOCK <lock>
    (SHARED-WRITE <counter> (+ (SHARED-READ <counter>) delta))))
```

Locks can also be constructed from shared queues, as is done by Lamina to implement mutual exclusion locks. To release the lock, a process places a token reference on the queue. A process acquires the lock by removing the token—any other process which attempts to remove it will be blocked until the owner of the lock replaces the token. Alternatively, reading but not removing the token (by using `shared-queue-top`) allows more than one process to be resumed. This last approach more closely resembles the type of synchronization provided by signalling and waiting on condition variables in a monitor.

Figure 5 shows an example of using some of these synchronization schemes in generating a closure to perform operations on a shared buffer realized as a shared variable array⁴. Processes first gain access to the shared array by spinning on a lock. Once access is

²In the current implementation, only FIFO queues are provided, and (in order to maintain a consistent timing model for cross address space transmissions) only shared variable or shared queue references may be placed on a shared queue.

³By convention, references to shared variables and shared queues are denoted by enclosing angle brackets, as in `<lock>`.

⁴The astute reader will note that the closure environment itself is not explicitly represented as shared; this is a modelling convenience due to the fact that the environment is not modified during the lifetime of the closure.

granted, items are inserted or removed. An attempt to put information in a full buffer returns nil. When an attempt is made to remove data from an empty buffer, a shared queue (rather than data) is returned; the requesting process may then wait for something to be placed on this queue by executing shared-queue-top.

```
(defun SHARED-BUFFER (size)
  (let ((<signal> (SHARED-QUEUE ':signal))
        (<empty> (SHARED-VARIABLE t))
        (<lock> (SHARED-VARIABLE t))
        (<buffer> (SHARED-ARRAY size :initial-element nil))
        (<head> (SHARED-VARIABLE 0))
        (<tail> (SHARED-VARIABLE 0)))
    #'(lambda (operation &optional value)
        (WITH-SPIN-LOCK <lock>
          (let* ((head (SHARED-READ <head>))
                 (tail (SHARED-READ <tail>)))
            (ecase operation
              (:insert
               (let ((new-tail (mod (1+ tail) size)))
                 (if (= head new-tail)
                     nil
                     (progn
                      t
                      (SHARED-ASET value <buffer> tail)
                      (when (SHARED-READ <empty>)
                        (SHARED-WRITE <empty> nil)
                        (SHARED-ENQUEUE <signal> <signal>))
                      (SHARED-WRITE <tail> new-tail))))))
              (:remove
               (if (not (= head tail))
                   (let ((new-head (mod (1+ head) size)))
                     (SHARED-WRITE <head> new-head)
                     (SHARED-AREF <buffer> head))
                   (unless (SHARED-READ <empty>)
                     (SHARED-WRITE <empty> t)
                     (SHARED-DEQUEUE <signal>))))))))))
```

Figure 5. *LAMINA Shared-variable synchronization mechanisms used to define a shared buffer.*

6.5. Shared Variable Matrix Manipulation

Now, let us consider the same sort of problem as we have already shown in serial, object-oriented and functional-programming forms being addressed with the use of the shared variable Lamina constructs. In this example we will have our matrix of values in the system's shared memory. We will create one process for each stage in the process ($f(x)$, $g(x)$, $print(x)$) and will make these processes pipe-line their execution. To do this we will use a pair of auxiliary tables, a lock table with one lock for each cell in the matrix being processed and a status table, which will specify for each cell in the matrix what point it has got to in the computation. For example, when the status table contains the token :F for a particular cell, we know that the F operation is now legal on this cell.

First, we define the appropriate versions of the f , g and $print$ functions. Each of these waits to grab the lock on the particular cell and, having got it, updates the matrix and the status table so that the next stage can take over.

```

(defun f-if (matrix lock-table status-table i j)
  ;; Applies the function F to a cell in the matrix after
  ;; grabbing the lock for that cell. It then updates the status
  ;; table so that the print operation is legal.
  (with-spin-lock (shared-aref lock-table i j)
    (setf (shared-aref matrix i j) (f (shared-aref matrix i j)))
    (setf (shared-aref status-table i j) :print)
  )
)

(defun g-if (matrix lock-table status-table i j)
  ;; Applies the function G to a cell in the matrix after grabbing
  ;; the lock for that cell. It then updates the status table so
  ;; that the F operation is legal.
  (with-spin-lock (shared-aref lock-table i j)
    (setf (shared-aref matrix i j) (g (shared-aref matrix i j)))
    (setf (shared-aref status-table i j) :f)
  )
)

(defun print-if (matrix lock-table status-table i j)
  ;; Applies the function Print-Out-Result to a cell in the
  ;; matrix after grabbing the lock for that cell. It then
  ;; updates the status table so that the G operation is once
  ;; more legal.
  (with-spin-lock (shared-aref lock-table i j)
    (print-out-result i j (shared-aref matrix i j))
    (setf (shared-aref status-table i j) :g)
  )
)

```

We now define the functions that are to be executed at the top level of each of the processes. We have one that applies the F function (f-ifier), one that prints and a G-ifier. In these functions we step through the matrix waiting until the appropriate status code appears. When we have the right status code for a particular cell we update the cell.

```

(defun f-ifier (matrix lock-table status-table)
  ;; The top level function for the process that performs
  ;; the F function.
  (loop for i from 0 below width ;; Loop over rows
    do (loop for j from 0 below height ;; Loop over columns
      ;; Loop until we get the right status code.
      ;; Dismiss is called so that we can be preempted
      ;; by any other processes on this site.
      do (loop until
        (equal :f (shared-aref status-table i j))
        do (dismiss)
      )
      ;; Perform the F transformation to the
      ;; matrix cell.
      (f-if matrix lock-table status-table i j)
    )
  )
)

```

```

(defun g-ifier (matrix lock-table status-table)
  ;; The top level function for the process that performs
  ;; the G function.
  (loop for i from 0 below width ;; Loop over rows
    do (loop for j from 0 below height ;; Loop over columns
      ;; Loop until we get the right status code.
      ;; Dismiss is called so that we can be preempted
      ;; by any other processes on this site.
      do (loop until
        (equal :g (shared-aref status-table i j))
        do (dismiss)
      )
      ;; Perform the G transformation to the
      ;; matrix cell.
      (g-ify matrix lock-table status-table i j)
    )
  )
)

(defun print-ifier (matrix lock-table status-table)
  ;; The top level function for the process that prints out
  ;; the results.
  (loop for i from 0 below width ;; Loop over rows
    do (loop for j from 0 below height ;; Loop over columns
      ;; Loop until we get the right status code.
      ;; Dismiss is called so that we can be preempted
      ;; by any other processes on this site.
      do (loop until
        (equal :print (shared-aref status-table i j))
        do (dismiss)
      )
      ;; Print out the result for this cell.
      (print-ify matrix lock-table status-table i j)
    )
  )
)

```

Now we can define the functions that will create and initialize the three shared matrices.

```

(defun make-shared-example-matrix ()
  ;; Create a 2 dimensional matrix that is initialized
  ;; with the numbers 0..8.
  (let ((matrix (shared-array dimensions)))
    (loop for i from 0 below width
      do (loop for j from 0 below height
        do (setf (shared-aref matrix i j)
          (+ (* i 3) j)
        )
      )
    )
    ;; Return the matrix
    matrix
  )
)

```

```

(defun make-shared-lock-table ()
  ;; Create a 2 dimensional matrix of locks.
  (let ((matrix (shared-array dimensions)))
    (loop for i from 0 below width
          do (loop for j from 0 below height
                  do (setf (shared-aref matrix i j)
                          (shared-variable t)
                          )
                  )
          )
    )
  ;; Return the matrix
  matrix
)

(defun make-shared-status-table ()
  ;; Creates the shared status table.
  (shared-array '(3 3) :initial-element :g)
)

```

Now we define the function that launches the three processes and creates the three shared matrices.

```

(defun start-shared-pipe ()
  ;; Starts up the shared-variable pipe-line process.
  ;; First it creates the shared matrices described above each
  ;; in a random shared memory.
  ;; Then it mounts the three processes executing the
  ;; F, G and Print operations.
  (let ((matrix
        (in-memory (RANDOM-MEMORY)
                   (make-shared-example-matrix)
                   )
        )
        (lock-table
        (in-memory (RANDOM-MEMORY) (make-shared-lock-table))
        )
        (status-table
        (in-memory (RANDOM-MEMORY) (make-shared-status-table))
        )
        )
    (mounting
     #'(lambda () (g-ifier matrix lock-table status-table))
     on (RANDOM-SITE)
    )
    (mounting
     #'(lambda () (f-ifier matrix lock-table status-table))
     on (RANDOM-SITE)
    )
    (mounting
     #'(lambda () (print-ifier matrix lock-table status-table))
     on (RANDOM-SITE)
    )
  )
)

```

```

(defun mount-start-shared-pipe ()
  ;; The top level function used to start things off.
  ;; This is needed because the CARE system requires that shared
  ;; memory be allocated by a mounted (fully-fledged) process.
  (mounting #'start-shared-pipe on (RANDOM-SITE))
)

(defun boot-shared-pipe-example ()
  ;; Boot the shared-pipe example on processing element (1 1)
  ;; at time = 0.
  (BOOT (at 0 '(1 1)
    ;; The following actually executes the example.
    (mount-start-shared-pipe)
  )
)
)

(boot-shared-pipe-example)

```

7. Conclusions

In this paper we have presented Lamina, a powerful programming model that allows the development and investigation of the performance of concurrent software using multiple programming metaphors, namely Object-Oriented, Shared-Variable and Functional-Programming styles.

We have shown that each of these three popular programming models can be implemented with the use of a single underlying system construct, the *stream*, which can be used either as a communication medium for a message passing programming style or to implement support for programming styles that use "promises" for (possibly infinite sequences of) values.

Lamina has been used extensively in the development of numerous programs, particularly in developing real-time concurrent expert system applications as part of the Advanced Architectures Project.

8. Bibliography

- [Andrews 83] Gregory R. Andrews and Fred B. Schneider. Concepts and Notations for Concurrent Programming. *Computing Surveys* 15, 1, pp 3-43. March 1983.
- [Byrd 87] Gregory T. Byrd, Russel T. Nakano, Bruce A. Delagi. *A Dynamic, Cut-Through Communications Protocol with Multicast*. Technical Report STAN-CS-87-1178 (KSL-87-44), Heuristic Programming Project, Computer Science Department, Stanford University, 1987.
- [Delagi 88] Bruce A. Delagi, Nakul P. Saraiya, Gregory T. Byrd and Sayuri Nishimura. *CARE User's Manual*. Technical Report KSL-88-53, Heuristic Programming Project, Computer Science Department, Stanford University, 1988.
- [Friedman 80] Daniel P. Friedman and David S. Wise. *An Indeterminate Constructor For Applicative Programming*. 7th Annual Symposium on Principles of Programming Languages, 1980, 245-250.

- [Halstead 85] Robert H. Halstead, Jr. *Multilisp: A Language for Concurrent Symbolic Computation*. ACM Transactions on Programming Languages and Systems 7, 4. October 1985.
- [Hewitt 73] Hewitt, C., P. Bishop and R. Steiger. *A Universal, Modular Actor Formalism for Artificial Intelligence*. Proceedings of the 3rd International Joint Conference on Artificial Intelligence: 235-245, 1973.
- [Lieberman 81] Henry Lieberman. *Thinking About Lots of Things Without Getting Confused*. AI Memo 626, MIT 1981.
- [Rice 88] James Rice. *The Advanced Architectures Project*. Technical Report KSL-88-71, Heuristic Programming Project, Computer Science Department, Stanford University, March 1989, also in AI Magazine Winter 1989, Vol. 11, No. 4 pages 26-39.
- [Shapiro 86] Ehud Shapiro. *Concurrent Prolog: A Progress Report*. Computer 18. August 1986, 44-58.
- [Steele 84] Guy L. Steele Jr., *Common Lisp the Language*, Digital Press, Burlington, MA, 1984.
- [Steele 76] Guy L. Steele Jr. *Lambda, the Ultimate Declarative*. AI Memo 379, MIT, November 1976.
- [Weinreb 81] Daniel Weinreb and David Moon. *Lisp Machine Manual*. Symbolics, Cambridge, MA 1981.

**Knowledge Systems Laboratory
Report No. KSL-86-22**

March 1986

The CAOS System

Eric Schoen

**Department of Computer Science
Stanford University
Stanford, CA 94305**

Abstract

The CAOS system is a framework designed to facilitate the development of highly concurrent real-time signal interpretation applications. It explores the potential of multiprocessor architectures to improve the performance of expert systems in the domain of signal interpretation.

CAOS is implemented in Lisp on a (simulated) collection of processor-memory sites, linked by a high-speed communications subsystem. The "virtual machine" on which it depends provides remote evaluation and packet-based message exchange between processes, using virtual circuits known as *streams*. To this presentation layer, CAOS adds (1) a flexible process scheduler, and (2) an object-centered notion of *agents*, dynamically-instantiable entities which model interpreted signal features.

This report documents the principal ideas, programming model, and implementation of CAOS. A model of real-time signal interpretation, based on replicated "abstraction" pipelines, is presented. For some applications, this model offers a means by large numbers of processors may be utilized without introducing synchronization-necessitated software bottlenecks.

The report concludes with a description of the performance of a large CAOS application over various sizes of multiprocessor configurations. Lessons about problem decomposition grain size, global problem solving control strategy, and appropriate services provided to CAOS by the underlying architecture are discussed.

Chapter 1

Introduction and Overview

This report documents the CAOS system, a portion of a recent experiment investigating the potential of highly concurrent computing architectures to enhance the performance of expert systems. The experiment focuses on the migration of a portion of an existing expert system application from a sequential uniprocessor environment to a parallel multiprocessor environment.¹

The application, called ELINT, is a portion of a multi-sensor information fusion system, and was written originally in AGE[2], an expert system development tool based on the blackboard paradigm. For the purposes of this experiment, ELINT was reimplemented in CAOS, an experimental concurrent blackboard framework based on the explicit exchange of messages between blackboard agents.

CAOS, in turn, relies on services provided by the underlying machine environment. In the present set of experiments, the environment is a simulation of a concurrent architecture, called CARE [5]. CARE simulates a square grid of processing nodes, each containing a Lisp evaluator, private memory, and a communications subsystem; message-passing is the only means of interprocessor communication.

CAOS is principally an operating system, controlling the creation, initialization, and execution of independent computing tasks in response to messages received from other tasks. Figure 1.1 illustrates the relationship between the various software components of the experiment.

The following chapter briefly describes the salient features of the CARE environment. Chapter 3 discusses the ideas behind the CAOS framework. Chapter 4 summarizes the CAOS programming environment, and Chapter 5 describes its implementation. The final chapter details the results of our experiments. Finally, Appendix A contains a simple CAOS example, and Appendix B presents a detailed, low-level look at the implementation of CAOS.

¹ This research was supported by DARPA Contract F30602-85-C-0012, NASA Ames Contract NCC 2-220-S1, and Boeing Contract W266875. Eric Schoen was supported by a fellowship from NL Industries.

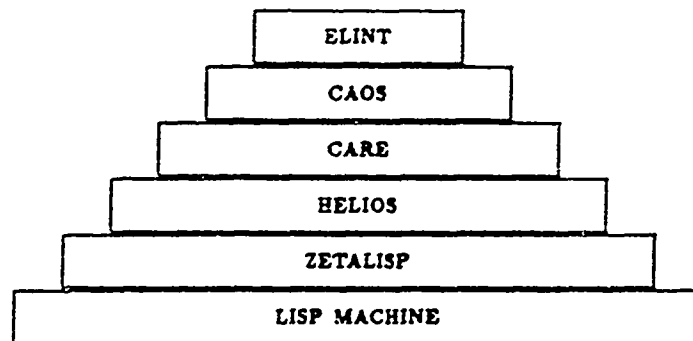


Figure 1.1: The relationship between ELINT, CARE, and CAOS

Chapter 2

An Overview of CARE

CARE is a highly-parameterized and well-instrumented multiprocessor simulation testbed, designed to aid research in alternative parallel architectures. It runs executes within Helios, a hierarchical, event-driven simulator which has been described elsewhere [3].

A typical CARE architecture is a grid of processing sites, interconnected by a dedicated communications network. For example, the research discussed in this paper was performed on square arrays of hexagonally connected processors (e.g., each processor is connected to six of its eight nearest neighbors, excluding processors at the edges of the grid).

Each processing site consists of an *evaluator*, a general-purpose processor/memory pair, and an *operator*, a dedicated communications and process scheduling processor which shares memory with the evaluator. Application-level computations take place in the evaluator, a component which is treated as a "black box" Lisp processor. No portion of its interior is simulated; the host Lisp machine serves as the evaluator in each processing site. The operator performs two duties. As a communications processor, it is responsible for routing messages between processing sites. As a scheduling processor, it queues application-level processes for execution in the evaluator (we discuss the scheduling mechanism in greater detail below). The operator is simulated and instrumented in great detail.

CARE allows a number of parameters of the processor grid to be adjusted. Among these parameters are: the speed of the evaluator, the speed of the communications network, and the speed of the process-switching mechanism. By altering these parameters, a single processor grid specification can be made to simulate a wide variety of actual multiprocessor architectures. For example, we can experiment with the optimal level-of-granularity of problem decomposition by varying the speed of both process-switching and communications.

Finally, CARE provides detailed displays of such information as evaluator, operator, and communication network utilization, and process scheduling latencies. This instrumentation package informs developers of CARE applications of how efficiently their systems make use of the simulated hardware.

2.1 The CARE Programming Model

CARE programs are made up of processes which communicate by exchanging messages. Messages flow across streams, virtual circuits maintained by CARE. The following services are used by CAOS:

New Process: Creates a new process on a specified site, running a specified top-level function. A new stream is returned, enabling the "parent" of the process to communicate with its "child." Pointers to the stream may be exchanged freely with other known processes on other sites.

New Stream: Creates a new stream whose target is the creating process.

Post Packet: Sends a message across a specified stream to a remote process.

Accept Packet: Returns the next message waiting on a specified stream. If no message is waiting when this operation is invoked, the invoking process is suspended and moved into the operator to await the arrival of a message.

Memory in each processing site is private. Ordinarily, intra-memory pointers may not be exchanged with processes in other sites. However, any pointer may be encapsulated in a remote-address, and may then be included in the contents of a message between sites. A remote address does not permit direct manipulation of remote structures; instead, it allows a process in one site to produce a local copy of a structure in another site.

Scheduling on a CARE node is entirely cooperative, and is based on message-passing. The message exchange primitives post-packet and accept-packet form the basis of process scheduling. A process wishing to block (yield control of the evaluator) does so by calling accept-packet to wait for a packet to arrive on a stream. The application program's scheduler awakens the process by calling post-packet to send a packet to the stream. The process is placed on the queue of processes waiting for the evaluator, and eventually regains control. The CAOS scheduler, which we describe in Section 5.3, is implemented in terms of this paradigm.

Chapter 3

The CAOS Framework

CAOS is a framework which supports the execution of multi-processor expert systems. Its design is predicated on the belief that future parallel architectures will emphasize limited communication between processors rather than uniformly-shared memory. We expected such an architecture would favor coarse-grained problem decomposition, with little or no synchronization between processors. CAOS is intended for use in real-time data interpretation applications, such as continuous speech recognition, passive radar and sonar interpretation, etc [7,11].

A CAOS application consists of a collection of communicating *agents*, each responding to a number of application-dependent, predeclared messages. An agent retains long-term local state. Furthermore, an arbitrary number of processes may be active at any one time in a single agent.

Whereas the uniprocessor blackboard paradigm usually implies pattern-directed, demon-triggered knowledge source activation, CAOS requires explicit messaging between agents; the costs of automatically communicating changes in the blackboard state, as required by the traditional blackboard mechanism, could be prohibitively expensive in the distributed-memory multiprocessor environment. Thus, CAOS is designed to express parallelism at a very coarse grain-size, at the level of knowledge source invocation in a traditional uniprocessor blackboard system. It supports no mechanism for finer-grained concurrency, such as within the execution of agent processes, but neither does it rule it out. For example, we could easily imagine the *methods* which implement the messages being written in QLisp [8], a concurrent dialect of Common Lisp.

3.1 The Structure of CAOS Applications

A CAOS application is structured to achieve high degrees of concurrency in two principal manners: *pipelining* and *replication*. Pipelining is most appropriate for representing the flow of information between levels of abstraction in an interpretation system, replication provides means by which the interpretation system can cope with arbitrarily high data rates.

3.1.1 Pipelining

Pipelining is a common means of parallelizing tasks through a decomposition into a linear sequence of independent stages. Each stage is assigned to a separate processing unit, which receives the output from the previous stage and provides input to the next stage. Optimally, when the pipeline reaches a steady-state, each of its processors is busy performing its assigned stage of the overall task.

CAOS promotes the use of pipelines to partition an interpretation task into a sequence of interpretation stages, where each stage of the interpretation is performed by a separate agent. As data enters one agent in the pipeline, it is processed, and the results are sent to the next agent. The data input to each successive stage represents a higher level of abstraction.

Advantages of Pipelining

Sequential decomposition of a large task is frequently very natural. Structures as disparate as manufacturing assembly lines and the arithmetic processors of high-speed computing systems are frequently based on this paradigm.

Pipelining provides a mechanism whereby concurrency is obtained without duplication of mechanism (that is, machinery, processing hardware, knowledge, etc). In an optimal pipeline of n processing elements, element 1 is performing work on task $t + n - 1$ when element 2 is working on task $t + n - 2$, and so on, such that element n is working on task t . As a result, the throughput of the pipeline is n times the throughput of a single processing element in the pipeline.

In the case of CAOS applications, the individual agents which compose an interpretation "pipeline" are themselves simple, but the overall combination of agents may be quite complex.

Disadvantages of Pipelining

Unfortunately, it is often the case that a task cannot be decomposed into a simple linear sequence of subtasks. Some stage of the sequence may depend not only on the results of its immediate predecessor, but also on the results of more distant predecessors, or worse, some distant successor (e.g., in feedback loops). An equally disadvantageous decomposition is one in which some of the processing stages take substantially more time than others. The effect of either of these conditions is to cause the pipeline to be used less efficiently. Both these conditions may cause some processing stages to be busier than others; in the worst case, some stages may be so busy that other stages receive no work at all. As a result, the n -element pipeline achieves less than an n -times increase in throughput. We discuss a possible remedy for this situation in the following section.

3.1.2 Replication

Concurrency gained through replication is ideally orthogonal to concurrency gained through pipelining. Any size processing structure, from individual processing elements to entire pipelines, is a candidate for replication. Consider a task which must be performed on average in time t , and a processing structure which is able to perform the task in time T , where $T > t$. If this task were actually a single stage in a larger pipeline, this stage would then be a bottleneck in the throughput of the pipeline. However, if the single processing structure which performed the task were replaced by

T/t copies of the same processing structure, the effective time to perform the task would approach t , as required.

Advantages of Replication

The advantages of replicating processing structure to improve throughput should be clear; n times the throughput of a single processing structure is achieved with n times the mechanism. Replication is more costly than pipelining, but it apparently avoids problems associated with developing a pipelined decomposition of a task.

Disadvantages of Replication

Our work leads us to believe that such replicated computing structures are feasible, but not without drawbacks. Just as performance gains in pipelines are impacted by inter-stage dependencies, performance gains in replicated structures are impacted by inter-structure dependencies.

Consider a system composed of a number of copies of a single pipeline. Further, assume the actions of a particular stage in the pipeline affects each copy of itself in the other pipelines. In an expert system, for example, a number of independent pieces of evidence may cause the system to draw the same conclusion, the system designer may require that when a conclusion is arrived at independently by different means, some measure of confidence in the conclusion is increased accordingly. If the inference mechanism which produces these conclusions is realized as concurrently-operating copies of a single inference engine, the individual inference engines will have to communicate between themselves to avoid producing multiple copies of the same conclusions. A stringent consistency requirement between copies of a processing structure decreases the throughput of the entire system, since a portion of the system's work is dedicated to inter-system communication.

3.2 An Example

We close this chapter by describing the organization of ELINT, illustrating the benefits and drawbacks of the CAOS framework applied to this problem. ELINT is an expert system whose domain is the interpretation of passively-observed radar emissions. Its goal is to correlate a large number of radar observations into a smaller number of individual signal emitters, and then to correlate those emitters into a yet smaller number of clusters of emitters. ELINT is meant to operate in real time; emitters and clusters appear and disappear during the lifetime of an ELINT run. The basic flow of information in ELINT is through a pipeline of the various agent types, which we now describe in detail.

Observation Reader

The observation reader is an artifact of the simulation environment in which ELINT runs. Its purpose is to feed radar observations into the system. The reader is driven off a clock; at each tick (1 ELINT "time unit"), it supplies all observations for the associated time interval to the proper observation handlers. This behavior is similar to that of a radar collection site in an actual ELINT setting.

Observation Handler

The observation handlers accept radar observations from associated radar collection sites (in the simulated system, the observations come from the observation reader agent). There may be a large number of observation handlers associated with each collection site. The collection site chooses to which of its many observation handlers to pass an observation, based on some scheduling criteria such as random choice or round-robin.

Each observation contains an externally-assigned number to distinguish the source of the observation from other known sources (the observation id is usually, but not always, correct). In addition, each observation contains information about the observed radar signal, such as its quality, strength, line-of-bearing, and operating mode. The observation does not contain information regarding the source's speed, flight path, and distance; ELINT will attempt to determine this information as it monitors the behavior of each source over time.

When an observation handler receives an observation, it checks the observation's id to see if it already knows about the emitter. If it does, it passes the observation to the appropriate emitter agent which represents the observation's source. If the observation handler does not know about the emitter, it asks an emitter manager to create a new emitter agent, and then passes the observation to that new agent.

Emitter Manager

There may be many emitter managers in the system. An emitter manager's task is to accept requests to create emitters with specified id numbers. If there is no such emitter in existence when the request is received, the manager will create one and return its "address" to the requesting observation handler. If there is such an emitter in existence when the request is received, the manager will simply return its address to the requestor. This situation arises when one observation handler requests an emitter than another observation handler had previously requested.

The reason for the emitter manager's existence is to reduce the amount of inter-pipeline dependency with respect to the creation of emitters. When ELINT creates an emitter, it is similar to a typical expert system's drawing a conclusion about some evidence; as discussed above, ELINT must create its emitters in such a way that the individual observation handlers do not end up each creating copies of the same emitter. Consider the following strategies the observation handlers could use to create new emitters:

1. The handlers could create the emitters themselves immediately. Since the collection site may pass observations with the same id to each observation handler, it is possible for each observation handler to create its own copy of the same emitter. We reject this method.
2. The handlers could create the emitters themselves, but inform the other handlers that they've done this. This scheme breaks down when two handlers try simultaneously to create the same emitter.
3. The handlers could rely on a single emitter manager agent to create all emitters. While this approach is safe from a consistency standpoint, it is likely to be impractical, as the single emitter manager could become a bottleneck in the interpretation.

4. The handlers could send requests to one of many emitter managers, chosen by some arbitrary method. This idea is nearly correct, but does not rule out the possibility of two emitter managers each receiving creation requests for the same emitter.
5. The handlers could send requests to one of many emitter managers, chosen through some algorithm which is invariant with respect to the observation id. This is in fact the algorithm in use in ELINT. The algorithm for choosing which emitter manager to use is based on a many-to-one mapping of observation id's to emitter managers.¹

Emitters

Emitters hold some state and history regarding observations of the sources they represent. As each new observation is received, it is added to a list of new observations. On a regular basis, the list of new observations is scanned for interesting information. In particular, after enough observations are received, the emitter may be able to determine its heading, speed, and location. The first time it is able to determine this information, it asks a cluster manager to either *match* the emitter to an old cluster or *create* a new cluster to hold the single emitter. Subsequently, it sends an update message to the cluster to which it belongs, indicating its current course, speed, and location.

Emitters maintain a qualitative confidence level of their own existence (*possible*, *probable*, and *positive*). If new observations are received often enough, the emitter will increase its confidence level until it reaches *positive*. If an observation is not received in the expected time interval, the emitter lowers its confidence by one step. If the confidence falls below *possible*, the emitter "deletes" itself, informing its manager, and any cluster to which it is attached.

Cluster Managers

The cluster managers play much the same role in the creation of cluster agents as the emitter managers play in the creation of emitters. However, it is not possible to compute an invariant to be used as a many-to-one mapping between emitters. If ELINT were to employ multiple cluster managers, the best strategy for choosing which of the many managers would still result in the possible creation of multiple instances of the "same" cluster. Thus, we have chosen to run ELINT with a single cluster manager. Fortunately, cluster creation is a rare event, and the single cluster manager has never been a processing bottleneck.

As indicated above, requests from emitters to create clusters are specified as match requests over the extant clusters. Emitters are matched to clusters on the basis of their location, speed, and heading. However, the cluster manager does not itself perform this matching operation. Although it knows about the existence of each cluster it has created, it does not know if the cluster has changed course, speed, and/or direction since it was originally created. Thus, the cluster manager asks each of its clusters to perform a match.

If either none of the clusters responds with a positive match, a new cluster is created for the emitter, if one cluster responds positively, the emitter is added to the cluster, and is so informed of this fact, if more than one cluster responds positively, an error (or a mid-air collision) must have occurred.

¹ The algorithm computes the observation id modulo the number of emitter managers, and maps that number to a particular manager.

Clusters

The radar emissions of clusters of emitters often indicates the actual behavior of the cluster. Cluster agents, therefore, apply heuristics about radar signals to determine whether the behaviors of the clusters they represent are threatening or not. This information, along with the course parameters of each radar source, is the "output" of the ELINT system. A cluster will delete itself if all constituent emitters have been deleted.

Chapter 4

Programming in the CAOS Framework

CAOS is package of functions on top of Lisp. These functions are partitioned into three major classes:

- Those which declare agents.
- Those which initialize agents.
- Those which support communication between agents.

We now describe the CAOS operators for each of these classes.

4.1 Declaration of agents

Agents are declared within an inheritance network. Each agent inherits the characteristics of its (multiple) parents. The simplest agent, *vanilla-agent*, contains the minimal characteristics required of a functional CAOS agent. All other CAOS agents reference *vanilla-agent* either directly or indirectly. Another predeclared agent, *process-agenda-agent*, is built on top of *vanilla-agent*, and contains a priority mechanism for scheduling the execution of messages.

Application agents are declared by augmenting the following characteristics of the base or other ancestral agents:

Local Variables: An agent may refer freely to any variable declared local. In addition, each local variable may be declared with an initial value.

Messages: The only messages to which an agent may respond are those declared in this table. This simplifies the task of a resource allocator, which must load application code onto each CARE site.

```
(defagent agent-name (parent1 ... parentn)
  (localvars variable1 ... variablen)
  (messages message1 ... messagen)
  (symbolically-referenced-agents agent1 ... agentn))
```

Figure 4.1: The basic form of `defagent`

Symbolically Referenced Agents: Some agents exist throughout a CAOS run. We call such agents *static*, and we allow code in agent message handlers to reference such agents by name. Before an agent begins running, each symbolic reference is resolved by the CAOS runtimes.

There are a number of additional characteristics; most of these are used by CAOS internally, and we will document these in the next chapter.

The basic form for declaring a CAOS agent is `defagent`. It has the form illustrated by Figure 4.1. The first element in each sublist is a keyword; there are a number of defined keywords, and their use in an agent declaration is strictly optional. An agent inherits the union of the keyword values of its parents for any unspecified keyword. Of those keywords which are specified, some are combined with the union of the keyword values of the agent's parents, and others supersede the values in the parents. Figure 4.2 contains the declaration of the emitter agent, one of the most complex examples in ELINT.

As we discuss in the next chapter, `defagent` forms are translated by CAOS into Flavors `defflavor` forms [4]. CAOS messages are then defined using the `defmethod` function of ZETALISP. These methods are free to reference the local variables declared in the `defagent` expression.

4.2 Initialization of agents

The initial CAOS configuration is specified by the `caos-initialize` operator, which takes the form illustrated by figure 4.3; for example, figure 4.4 is ELINT's initialization form.

The first portion of the form creates the static agents. In figure 4.4, a static agent named `el-gotcha-handler-1`, an instance of the class `el-observation-handler`, is created on the CARE site at coordinates (1,2) in the processor grid.

The second portion of the form is a list of LISP expressions to be evaluated sequentially when CAOS's initialization phase is complete. Each expression is intended to send a message to one of the static agents declared in the first part of the form. These messages serve to initialize the application; in figure 4.4, the initialization messages open log files and start the processing of ELINT observations.

Agents may also be created dynamically. The `create-agent-instance` function accepts an agent class name and a location specification;¹ the `remote-address` of the newly-created agent is returned. While dynamically created agents may not be referenced symbolically, their `remote-address`'s may be exchanged freely.

¹Currently, agents may be created at or near specified CARE sites. CAOS makes no attempt at dynamic load balancing.

```

(defagent el-emitter (process-agenda-agent)
  (localvars
    (process-agenda '(el-undo-collection-id-error
                      el-change-cluster-association
                      el-emitter-update-on-time-tick
                      el-initialize-emitter
                      el-update-emitter-from-observation))
    (last-observed -1000000)
    (cluster-manager 'cluster-manager-0)
    manager
    id
    type
    observed
    fires
    last-heading
    last-mode
    confidence
    cluster
    new-observations-since-time-tick-flag
    id-errors
    gc-flag)
  (messages
    el-update-emitter-from-observation
    el-initialize-emitter
    el-change-cluster-association
    el-undo-collection-id-error)
  (symbolically-referenced-agents
    el-collection-reporter-0
    el-correlation-reporter-0
    el-threat-reporter-0
    el-cluster-manager-0
    el-cluster-manager-1
    el-cluster-manager-2
    el-big-ear-handler
    el-gotcha-handler
    el-emitter-trace-reporter-0))

```

Figure 4.2: The emitter agent


```
(caos-initialize
  ((agent - name1 agent - class site - address)
   ...)
  ((initial - message1)
   ...))
```

Figure 4.3: The basic CAOS initialization form

4.3 Communications Between Agents

Agents communicate with each other by exchanging messages. CAOS does not guarantee that messages reach their destinations: due to excessive message traffic or processing element failure, messages may be delayed or lost during routing. It is the responsibility of the application program to detect and recover from lost messages. Commensurate with the facilities provided by CARE, messages may be tagged with routing priorities; however, higher priority messages are not guaranteed to arrive before lower-priority messages sent concurrently.

Two classes of messages are defined: those which return values (called *value-desired messages*), and those which do not (called *side-effect messages*). The value-desired-messages are made to return their values to a special cell called a *future*. Processes attempting to access the value of a future are blocked until that future has had its value set. It is possible for the value of a future to be set more than once, and it is possible for there to be multiple processes awaiting a future's value to be set.²

4.3.1 Sending messages

The CARE primitive *post-packet*, which sends a packet from one process to another, is employed in CAOS to produce three basic kinds of message sending operations:

post: The *post* operator sends a side-effect message to an agent. The sending process supplies the name or pointer to the target agent, the message routing priority, the message name and arguments. The sender continues executing while the message is delivered to the target agent.

post-future: The *post-future* operator sends a value-desired message to the target agent. The sending process supplies the same parameters as for *post*, and is returned a pointer to the future which will eventually be set by the target agent. As for *post*, the sender continues executing while the message is being delivered and executed remotely.

A process may later check the state of the future with the *future-satisfied?* operator, or access the future's value with the *value-future* operator, which will block the process until the future has a value.

post-value. The *post-value* operator is similar to the *post-future* operator, however, the sending process is delayed until the target agent has returned a value. *post-value* is defined in terms of *post-future* and *value-future*.

²Futures were also used in QLisp and Multilisp [9]. The HEP Supercomputer [6] implemented a simple version of futures as a process synchronization mechanism.

```

(caos-initialize
  ((el-observation-reader-0 el-observation-reader (2 2))
   (el-big-ear-handler-1 el-observation-handler (1 1))
   (el-big-ear-handler-2 el-observation-handler (1 1))
   (el-gotcha-handler-1 el-observation-handler (1 2))
   (el-gotcha-handler-2 el-observation-handler (1 2))
   (el-emitter-manager-0 el-emitter-manager (2 1))
   (el-emitter-manager-1 el-emitter-manager (2 2))
   (el-collection-reporter-0 el-collection-reporter (1 2))
   (el-correlation-reporter-0 el-correlation-reporter (1 3))
   (el-threat-reporter-0 el-threat-reporter (1 3))
   (el-emitter-trace-reporter-0 el-emitter-trace-reporter
                                (3 2))
   (el-cluster-trace-reporter-0 el-cluster-trace-reporter
                                (3 1))
   (el-cluster-manager-0 el-cluster-manager (2 1)))
  ((post el-observation-reader-0 nil
        'el-open-observation-file
        *elint-data-files*)
   (post el-collection-reporter-0 nil
        'el-initialize-reporter t
        "elint:reports;collections.output")
   (post el-correlation-reporter-0 nil
        'el-initialize-reporter t
        "elint:reports;correlations.output")
   (post el-threat-reporter-0 nil
        'el-initialize-reporter t
        "elint:reports;threats.output")
   (post el-emitter-trace-reporter-0 nil
        'initialize-trace-reporter t
        "elint:reports;emitter.traces")
   (post el-cluster-trace-reporter-0 nil
        'initialize-trace-reporter t
        "elint:reports;cluster.traces"))))

```

Figure 4 4: The initialization declaration for ELINT.

4.3.2 Detecting Lost Messages

It is possible to detect the loss of value-desired messages by attaching a timeout to the associated future. The functions `post-clocked-future` and `post-clocked-value` are similar to their untimed counterparts, but allow the caller to specify a *timeout* and *timeout action* to be performed if the future is not set within the timeout period. Typical actions include setting the future's value with a default value, or resending the original message using the `repost` operator.

4.3.3 Sending to Multiple Agents

There exist versions of the basic posting operators which allow the same message to be sent to multiple agents.³ `multipost` sends a side effect message to a list of agents; `multipost-future` and `multipost-value` send a value-desired message to a list of agents. In the latter case, the associated future is actually a list of futures; the future is not considered set until all target agents have responded. The value of such a message is an association-list; each entry in the list is composed of an agent name or `remote-address` and the returned message value from that agent. There exist clocked versions of these functions (called, naturally, `multipost-clocked-future` and `multipost-clocked-value`) to aid in detecting lost multicast messages.

4.4 Communications Between Processes

Processes in each agent communicate using the shared local variables declared in the agent. Besides sharing previously computed results this way, processes may also share the results of ongoing computations.

Consider the following scenario: within an agent, some process is currently computing some answer. At the same time, another process begins executing, and realizes somehow that the answer it needs to compute is the same answer the other process is already computing. The second process could take one of two actions: it could continue computing the answer, even though this would mean redundant work, or it could wait for the first process to complete, and return its answer. The second approach is feasible, but it does tie up resources in the form of an idle process.

The CAOS operators `attach` and `my-handle` offer a third alternative solution. If a process knows it may ultimately produce an answer needed by more than one requesting agent, it obtains its "handle" (Section 5.4) by calling `my-handle`, and places it in a table for other processes to reference. Any other process wishing to return the same answer as the first calls `attach`, with the first process's handle as argument. The first process returns its answer to all requesting agents waiting for answers from the other processes, and the other processes return no value at all.

4.5 What CAOS Offers Over CARE

CAOS is a large system. It is reasonable to ask what advantages there are to programming in CAOS as opposed to programming in CARE. We believe there are three major advantages:

³Neither CAOS nor CARE currently support a *predicated multicast* mode, wherein messages would be sent to all agents satisfying a particular predicate; messages can only be sent to a fully-specified list of agents.

Clarity: The framework in which an agent is declared makes explicit its storage requirements and functional behavior. In addition, the agent concept is a helpful abstraction at which to view activity in a multiprocessing software architecture. The concept lets us partition a flat collection of processes on a site into groups of processes attached to agents on a site. CAOS guarantees the only interaction between processes attached to different agents is by message-passing.

Convenience: The programmer is freed from interfacing to CARE's low-level communications primitives. As we said earlier, CAOS is basically an operating system, and as such, it shields the programmer from the same class of details a conventional operating system does in a conventional hardware environment.

Flexibility: Currently, CARE schedules processes in a strict first-in, first-out manner. CAOS, on the other hand, can implement arbitrary scheduling policies (though at a substantial performance cost; we discuss this in Chapter 6).

Chapter 5

The Runtime Structure of CAOS

CAOS is structured around three principal levels: site, agent, and process. Two of these levels—site and process—reflect the organization of CARE; the remaining (agent) level is an artifact of CAOS. We discuss first the general design principles underlying CAOS, and then describe in greater detail the functions and structure of each of CAOS's levels. Appendix B offers a complete guide to the algorithms and data structures employed in CAOS.

5.1 General Design Principles

The implementation of CAOS described in this paper is written in ZETALISP, a dialect of Lisp which runs on a number of commercially available single-user Lisp workstations. ZETALISP includes an object-oriented programming tool, called Flavors, which has proved to be a very powerful facility for structuring large Lisp applications.

In Flavors, the behavior of an object is described by templates known as *classes*. An *instance*, a representation of an individual object, is created by instantiating a class. Instances respond to messages defined by their class, and contain static local storage in the form of *instance variables*. Classes are defined within an inheritance network; each instance contains the instance variables and responds to the messages defined in its class, as well as those of the classes from which its class inherits.

An appropriate usage for Flavors is the modelling of the behavior of objects in some (not necessarily real) world. For example, CAOS site and agents structures are realized as Flavors instances. The characteristics to be modelled are codified in instance variables and message names. In a well-designed application, messages and variables are consistently named, thus, the implementation of a particular behavior is totally encapsulated in the anonymous function which responds to a message.

5.1.1 Extending the Notion

In some sense, a Flavors instance is an abstract data type. The instance holds state, and provides advertised, public interfaces (messages) to functions which change or access its state. The internal data representation and implementations of the access functions are private.

In Flavors, the abstract data type notion is unavailable within an individual instance. Frequently, the individual instance variables hold complex structures (such as dictionaries and priority queues) which ought to be treated as abstract data types, but there exist no common means within the standard Flavors mechanism for doing so.

CAOS, however, supports such a mechanism, by providing a means of sending messages to instance variables (rather than to the instances themselves). The instance variables are thus able to store anonymous structures, which are initialized, modified, and accessed through messages sent to the variable. Similar mechanisms exist in the Unit Package [14] and in the STROBE system [13], both frameworks for representing structured knowledge.

The CAOS environment includes a number of abstract data types which were found to be useful in supporting its own implementation. The most commonly used are:

Dictionary: The dictionary is an association list. It responds to `put`, `get`, `add`, `forget`, and `initialize` messages.

Sorted Dictionary: The sorted-dictionary is also implemented as an association list, and responds to the same messages as does the standard dictionary. However, the sorted-dictionary invokes a user-supplied priority function to merge new items into the dictionary (higher-priority items appear nearer the front of the dictionary). This dictionary is able to respond to the `greatest` message, which returns the entry with the highest priority, and to the `next` message, which returns the entry with the next-highest priority as compared to a given entry.

The sorted-dictionary is used primarily to hold time-indexed data which may be collected out-of-order (e.g. when data for time $n + 1$ may arrive before data for time n).

Hash Dictionary: The hash-dictionary is implemented with a hash table, and responds to the same messages as the unsorted association list dictionary.

Queue. The queue data type is a conventional first-in, first-out storage structure. The `put` message enqueues an item on the tail of the queue, while the `get` message dequeues an item from the head of the queue.

Priority Queue. The priority-queue data type supports a dynamic heapsort, and is implemented as a partially-ordered binary tree. It responds to `put`, `get`, and `initialize` messages. Associated with the queue is a function which computes and compares the priority of two arbitrary queue elements; this function drives the rebalancing of the binary tree when elements are added or deleted.

Monitor. A monitor provides mutual exclusion within a dynamically-scoped block of Lisp code. It is similar in implementation to the monitors of Interlisp-D and Mesa [10].

If the monitor is unlocked, the `obtain-lock` message stores the caller's process id as the monitor's owner, and marks the monitor as locked; otherwise, if the monitor is locked, the `obtain-lock` message places the caller's process id on the tail of the monitor's waiting queue, and suspends the calling process.

The `release-lock` message removes the process id from the head of the monitor's waiting queue, marks the monitor's owner to be that id, and reschedules the associated process.

Monitors are normally accessed using the `with-monitor` form, which accepts the name of an instance variable containing a monitor, and which cannot be entered until the calling process obtains ownership of the monitor. The `with-monitor` form guarantees ownership of the monitor will be relinquished when the calling process leaves the scope of the form, even if an error occurs.

5.2 The CAOS Site Manager

The site manager consists of a `Flavors` instance containing information global to the site—information needed by all agents located on the site. In addition, the site manager includes a `CARE`-level process which performs the functions of creating new agents and translating agent names into agent addresses, as described below.

The following instance variables are part of the site manager:

incoming-stream: This instance variable contains the `CARE` input stream address on which the site manager process listens for requests. Agents needing to send messages to their site manager may reference this instance variable in order to discover the address to which to direct site requests.

static-agent-stream-table: This instance variable is a dictionary which maps agent names into the `CARE` streams which may be used to communicate with the agents. The entries in this dictionary reflect statically-created agents; new entries are added as the result of `new-initial-agent-online` messages directed to the site (see below). The dictionary is used to resolve agent name-to-address requests from agents created locally.

unresolved-agent-stream-table: The site manager keeps track of agent names it is not able to translate to addresses by placing unsatisfiable `request-symbolic-reference` requests in this dictionary. The keys of the dictionary are unresolvable agent names. As the agent names become resolvable, the unsatisfied requests are satisfied, and the corresponding entries are removed from the dictionary.

After the initialization phase of a `CAOS` application has completed, there will be no entries in this dictionary in any of the sites.

local-agents: This instance variable is a dictionary whose keys are the names of agents located on the site, and whose values are pointers to the `Flavors` instances which represent each agent. `local-agents` is used only for debugging and status-reporting purposes.

free-process-queue: When a `CARE` process which was created to service a request finishes its work, it tries to perform another task for the agent in which it was created. If the agent has no work to do, the process suspends itself, after enqueueing identifying information in this instance variable, which holds a queue abstract data type. When any agent on the same site needs a new process to service some request, it checks this queue first; if there are any suspended (free) processes waiting in this queue, it dequeues one and gives it a task to perform. If this queue is empty, the agent asks `CARE` to create a new process.

The site manager responds to the following messages:

new-initial-agent-online: As each static agent starts running during initialization of a CAOS run, it broadcasts its name and CARE input stream to every site in the system, using this message. The correspondence between the sending agent's name and address is placed in the **static-agent-stream-table** dictionary for future reference by agents located on the receiving sites. If any agents have placed requests for this new agent in the **unresolved-agent-stream-table**, messages containing the new agent's name and address are sent to the waiting agents.

request-symbolic-reference: Whenever a static agent is created, it runs an initialization function, which among other tasks, caches needed agent name-to-address translations. For each translation, the agent sends this message to its site manager. If the site manager can resolve the name upon receipt of the message, it responds immediately; otherwise, it queues the request in the **unresolved-agent-stream-table**, and defers answering until it is able to satisfy the request. The requesting agent waits until it has received the answer before requesting another translation.

make-new-agent: This message is sent to a site to cause a new agent to be created during the course of a CAOS run. The site manager creates the new (dynamic) agent and returns the agent's input stream to the sender of this message. The newly-created agent is *not* placed in the **static-agent-stream-table**; thus, the only way to advertise the existence of such a dynamically-created agent is by the creator of an agent passing the returned input stream to other agents.

5.3 The CAOS Agent

As discussed above, CAOS agents are implemented as Flavors instances. Their class definitions are defined by translating **defagent** expressions into **defflavor** expressions. CAOS itself defines two basic agent classes: **vanilla-agent** and **process-agenda-agent**. **vanilla-agent** defines the minimal agent; **process-agenda-agent** is defined in terms of **vanilla-agent**, but adds the ability to assign priorities to messages.¹ These basic agents are fully-functional, but lack domain-specific "knowledge," and cannot be used directly in problem solving applications.

As stated in the previous chapter, a CAOS agent is a multiple-process entity. Most of these processes are created in the course of problem-solving activity; we refer to these as *user processes*. At runtime, however, there are always two special processes associated with each CAOS agent. One of these processes monitors the CARE stream by which the agent is known to other agents. The other participates in the scheduling of user processes. We shall refer to the first of these processes as the agent *input monitor*, and to the second of these processes as the agent *scheduler*. We explain in detail the functioning of these two processes in the next section.

We describe here the role of important instance variables in a basic CAOS agent:

¹This is important for applications in which one agent must respond rapidly to a posting from another agent. Assigning a message a high priority will cause that message to be processed ahead of any other messages with lower priorities.

self-address: This instance variable is an analogue of Flavors' **self** variable. Whereas **self** is bound to the Flavors instance under which a message is executing, **self-address** is bound to the stream of the agent under which a CAOS message is executing. Thus, an agent can post a message to itself by posting the message to **self-address**.

runnable-process-stream: This instance variable points to the stream on which the scheduler process listens. Processes which need to inform the scheduler of various conditions do so by sending CARE-level messages to this stream.

running-processes: This variable holds the list of user processes which are currently executing within the agent. The current CARE architecture supports only a single evaluator on each site. CAOS tries to keep a number of user processes ready to execute at all times; thus, the single CPU is kept as busy as possible.

runnable-process-list: A priority queue containing the runnable user processes. As a process is entered on the queue, its priority is calculated to determine its ranking in the partial ordering. There are two available priority evaluation functions: the first computes the priority based solely on the time the process entered the system; the second considers the assigned priority of the executing message before considering the entry time of the process. These two functions are used to implement the scheduling algorithms of the **vanilla-agent** and the **process-agenda-agent**, respectively.

scheduler-lock: The scheduler data structures are subject to modification by any number of processes concurrently. The **scheduler-lock** is a monitor which provides mutual exclusion against simultaneous access to the scheduler database.

5.4 The CAOS Process

In this section, we describe the mechanism by which CAOS user processes are scheduled for execution on CARE sites. User processes are created in response to messages from other agents. Associated with each user process is a data structure called a **runnable-item**. The **runnable-item** contains the following fields:

message-name, -args, -id, -answer-targets: These fields store the information necessary to handle a message request and send the resulting answer back to the proper agents.

for-effect: This field is a boolean, and indicates whether the message is being executed for effect or value. This corresponds directly to the source of the message coming from a **post** operation or a **post-future** operation.

state: This field indicates the state of the process. The possible states that a process may enter, and the finite state machine which defines the state transition are discussed in the next section.

context: This field contains a pointer to the CARE stream upon which the process waits when it not runnable. A process (such as the scheduler) wishing to wake another process simply sends a message to this stream. The suspended process will thus be awakened (by CARE).

time-stamp: This field contains the time at which the process entered the system. It is used by the functions which calculate the execution priority of processes.

The CAOS scheduler's only handle on a process is the process's **runnable-item**. In fact, the only communication between a user process and the CAOS scheduler consists of the exchange of **runnable-item**'s.

5.5 Flow of Control

In the following, we detail how a user process, the CAOS input monitor, and the CAOS scheduler interact to process a message request from a remote agent. For purposes of exposition, we assume the following sequence of events:

1. An agent, **agent-1**, executes a post operation, with **agent-2** as the target. The posting is for the message named **message-a**.
2. **agent-2** receives and executes the posting. In order to complete the execution of **message-a**, it must perform a **post-value** operation to a third agent, **agent-3**.

We begin at the point where **agent-1** has performed its post operation.

5.5.1 Input Processing

The input monitor process handles requests and responses from remote agents. When the message from **agent-1** enters **agent-2**, its input monitor creates a new **runnable-item** to hold the state of the request. The message name, arguments, id, and answer targets are copied from the incoming message into the **runnable-item**. The **runnable-item**'s state is set to **never-run**, and its time stamp is set to the current time. In order to queue the message for execution, the input monitor takes one of two actions.

If the agent's **runnable-process-list** is empty, the **runnable-item** is sent in a message to the agent scheduler process (by sending the item in a message to the stream whose address is found in the agent's **runnable-process-stream** instance variable). When the agent's **runnable-process-list** is empty, the scheduler process is guaranteed to be waiting for messages sent to the scheduler stream, and hence, will be awakened by the message sent from the input monitor. The scheduler then computes the priority of the message, and places the **runnable-item** in its **runnable-process-list**.

If the agent's **runnable-process-list** is not empty, the input monitor computes the message's priority and places the **runnable-item** on the **runnable-process-list** itself. When the queue is not empty, it is guaranteed that the scheduler will examine the queue sometime in the future to make scheduling decisions, thus, it is not necessary to send any messages to the scheduler to inform it of the existence of new processes.

5.5.2 Creating Processes

Eventually, the newly-created `runnable-item` will reach the head of `agent-2's runnable-process-list`. At this time, there is still no process associated with the item, so the scheduler creates a process using the facilities of CARE, adds the process to the `running-processes` list, and passes it its `runnable-item`. The process will eventually gain control of the evaluator, and will set the state of its `runnable-item` to `running`. It then begins executing the requested posting.

5.5.3 Requesting Remote Values

At some point, the process executing on `agent-2` requires a value from `agent-3`, and performs a `post-value` operation to acquire it. The process looks up the address of `agent-3`, and posts a message which contains the appropriate message name, arguments, id, and answer target. The `message-id` unambiguously identifies the future upon which the process will be waiting for the value to be returned. The answer target is the agent's own `self-address`; when the answer is received by the input monitor process, it will be forwarded to the appropriate future, and the process will be reawakened.

In the meantime, the process sets its state to `suspended`, removes its `runnable-item` from the `running-processes` list, and appends it to the list of processes already waiting for the future to be satisfied. If the `runnable-process-list` is not empty, the suspending process wakes the process at the head of the queue.² The suspending process then waits for a message on its wakeup stream, the stream whose address is in the context field of its `runnable-item`.

5.5.4 Answer Processing

Some time later, `agent-3` will have completed its computations, and will have returned the desired answer to `agent-2`. The answer will be received by `agent-2's` input monitor process, which will recognize the input as a value to be placed in a future. The input monitor sets the value field of the appropriate future, and moves the `runnable-items` of the processes waiting on the future to the `runnable-process-list`.

If the queue was previously empty, the agent must have been (or will soon be) entirely idle; thus, the `runnable-items` are sent to the scheduler in a message, causing the scheduler to be reawakened. If the queue was not previously empty, the agent must be busy, so the items are simply added to the queue according to their priorities. In both cases, the `runnable-items` are placed in the `runnable` state.

5.5.5 Reawakening Suspended Processes

When the `runnable runnable-item` reaches the head of `agent-2's runnable-process-list`, a message (which contains no useful information) is sent to its associated process's wakeup stream. As a result, process eventually wakes up, gains control of the evaluator, and sets its state to `running`.

²In effect, the process takes on the role of the scheduler. Although the system would continue to work with only a designated scheduler process performing scheduler duties, this arrangement permits scheduling to take place with minimal latency. As a result, fewer evaluator cycles are wasted waiting for the scheduler process to run the next user process.

5.5.6 Completing Computation .

A process may perform any number of post, post-future, or post-value operations during its lifetime. Eventually, however, the process will complete, having computed a value which may or may not be sent back to the requesting agent. If the process was suspended for any portion of its lifetime, another process may have attached to it; in this case, the process may have more than one requesting agent to which to return an answer.

Before the process terminates, it examines the head of the runnable-process-list. If the queue is empty, the process simply goes away. If the runnable-item at the head of the queue is runnable, it sends the appropriate message to awaken the associated process. Finally, if the item is never-run, the process makes itself the process associated with this new runnable-item, and executes the new message in its own context.³ Barring this possibility, the process "queues" itself on a free process queue associated with the site manager; when a new process is needed by an agent on the site, one is preferentially removed from this queue and recycled before a entirely new process is created. This way, processes, which are expensive to create, are reused as often as possible.

³ This is another situation in which an application process performs scheduling duties.

Chapter 6

Results and Conclusions

The CAOS system we have described has been fully implemented and is in use by two groups within the Advanced Architectures Project. CAOS runs on the Symbolics 3600 family of machines, as well as on the Texas Instruments *Explorer* Lisp machine. ELINT, as described in Section 3.2, has also been fully implemented. We are currently analyzing its performance on various size processor grids and at various data rates.

6.1 Evaluating CAOS

CAOS is a rather special-purpose environment, and should be evaluated with respect to the programming of concurrent real-time signal interpretation systems. In this chapter, we explore CAOS's suitability along the following dimensions:

- Expressiveness
- Efficiency
- Scalability

6.1.1 Expressiveness

When we ask that a language be suitably *expressive*, we ask that its primitives be a good match to the concepts the programmer is trying to encode. The programmer shouldn't need to resort to low-level "hackery" to implement operations which ought to be part of the language. We believe we have succeeding in meeting this goal for CAOS (although to date, only CAOS's designers have written CAOS applications). Programming in CAOS is programming in Lisp, but with added features for declaring, initializing, and controlling concurrent, real-time signal interpretation applications.

6.1.2 Efficiency

CAOS has a very complicated architecture. The lifetime of a message, as described in Section 5.5, involves numerous processing states and scheduler interventions. Much of this complexity derives from the desire to support alternate scheduling policies within an agent. The cost of this complexity is approximately one order of magnitude in processing latency. For the common settings of simulation parameters, CARE messages are exchanged in about 2-3 milliseconds, while CAOS messages require about 30 milliseconds. It is this cost which forces us to decompose applications coarsely, since more fine-grained decompositions would inevitably require more message traffic.

We conclude that CAOS does not make efficient use of the underlying CARE architecture. A compromise, which we are just beginning to explore, would be to avoid the complex flow of control described in Section 5.5 in agents whose scheduling policies are the same as CARE's (FIFO). In such agents, we could reduce the CAOS runtimes to simple functional interfaces to CARE. We anticipate such an approach would be much more efficient.

6.1.3 Scalability

A system which scales well is one whose performance increases commensurately with its size. Scalability is a common metric by which multiprocessor hardware architectures are judged: does a 100-processor realization of a particular architecture perform 10 times better than a 10-processor realization of the same architecture? Does it perform 5 times better? Only just as well? Or *Worse*? In hardware systems, scalability is typically limited by various forms of contention in memories, busses, etc. The 100-processor system might be slower than the 10-processor system because all interprocessor communications are routed through an element which is only fast enough to support 10 processors.

We ask the same question of a CAOS application: does the throughput of ELINT, for example, increase as we make more processors available to it? This question is critical for CAOS-based real-time interpretation systems; our only means of coping with arbitrarily large data rates is by increasing the number of processors. Section 6.2 discusses this issue in detail.

We believe CAOS scales well with respect to the number of available processors. The potential limiting factors to increasing scaling are (1), increased software contention, such as inter-pipeline bottlenecks described in Section 3.1.2, and (2), increased hardware contention, such as overloaded processors and/or communication channels. Software contention can be minimized by the design of the application. Communications contention can be minimized by executing CAOS on top of an appropriate hardware architecture (such as that afforded by CARE). CAOS applications tend to be coarsely decomposed—they are bounded by computation, rather than communication—and thus, communications loading has never been a problem.

Unfortunately, processor loading remains an issue. A configuration with poor *load balancing*, in which some processors are busy, while others are idle, does not scale well. Increased throughput is limited by contention for processing resources on overloaded sites, while resources on unloaded sites go unused. The problem of automatic load balancing is not addressed by CAOS: agents are assigned to processing sites on a round-robin basis, with no attempt to keep potentially busy agents apart.

ELINT Performance Dimension	Control Type/Grid Size					
	NC	CC	CC	CT	CT	CT
	4 x 4	4 x 4	6 x 6	2 x 2	4 x 4	6 x 6
FALSE ALARMS	1	0	0	0	0	0
REINCARNATION	49	42	2	0	0	0
CONFIDENCE LEVEL	19	20	90	89	93	95
FIXES	48	42	99	100	100	100
FUSION	0	0	77	35	88	89

Table 6.1: Quality of ELINT performance of various grid sizes and control strategies (1 ELINT time unit = 0.1 seconds).

6.2 Evaluating ELINT Under CAOS

Our experience with ELINT indicates the primary determiner of throughput and answer-quality is the strategy used in making individual agents cooperate in producing the desired interpretation. Of secondary importance is the degree to which processing load is evenly balanced over the processor grid. We now discuss the impact of these factors on ELINT's performance.

The following three strategies were used in our experiments:

- NC: This strategy represents limited inter-agent control. No attempt is made to prevent concurrent creation of multiple copies of the "same" agent (this possibility arises when multiple requests to create the agent arrive simultaneously at a single manager). As a result, multiple, non-communicating copies of an abstraction pipeline are created; each receives a only portion of the input data it requires. The NC strategy was expected to produce poor results, and was intended only as a baseline against which to compare more realistic control strategies.
- CC: In this strategy, the manager agents assure that only one copy of a agent is created, irrespective of the number of simultaneous creation requests; all requestors are returned pointers to the single new agent. Originally, we believed the CC (for "creation control") strategy would be sufficient for ELINT to produce correct high-level interpretations.
- CT: The CT ("creation and time control") strategy was designed to manage skewed views of real-world time which develop in agent pipelines. In particular, this strategy prevents an emitter agent from deleting itself when it has not received a new observation in a while, yet some observation-handler agent has sent the emitter an observation which it has yet to receive.

Table 6.1 illustrates the effects of various control strategies and grid sizes. The table presents six performance attributes by which the quality of an ELINT run is measured.

False Alarms: This attribute is the percentage of emitter agents that ELINT should not have hypothesized as existing.

ELINT was not severely impacted by false alarms in any of the configurations in which it was run.

Control Type	Simulated Time (sec)		
	2 x 2	4 x 4	6 x 6
NC		> 11.19 ^a	
CC		10.87	5.12
CT	11.80	8.10	4.17

^aThis run was far from completion when it was halted due to excessive accumulated wall-clock time.

Table 6.2: Simulated time required to complete an ELINT run (1 ELINT time unit = 0.1 seconds).

Control Type	Message Count		
	2 x 2	4 x 4	6 x 6
NC		> 16118	
CC		7375	
CT	4516	4703	4616

Table 6.3: Number of messages exchanged during an ELINT run (1 ELINT time unit = 0.1 seconds).

GRID SIZE	1 x 1	2 x 2	3 x 3	4 x 4	5 x 5	6 x 6
SIMULATED TIME (sec)	9.42	3.20	1.49	0.74	0.52	0.56

Table 6.4: Overall Simulation Times for CT Control Strategy (1 ELINT time unit = 0.01 seconds, debugging agents turned off).

Reincarnation: This attribute is the percentage of recreated emitter agents (e.g., emitters which had previously existed but had deleted themselves due to lack of observations). Large numbers of reincarnated emitters indicate some portion ELINT is unable to keep up with the data rate (i.e., the data rate may be too high globally, so that all emitters are overloaded, or the data rate may be too high locally, due to poor load balancing, so that some subset of the emitters are overloaded).

The CT control strategy was designed to prevent reincarnations; hence, none occurred when CT was employed on any size grid. When CC was used, only the 6×6 grid was large enough for ELINT to keep up with the input data rate.

Confidence Level: This attribute is the percentage of correctly-deduced confidence levels of the existence of an emitter.

The correct calculation of confidence levels depends heavily on the system being able to cope with the incoming data rate. One way to improve confidence levels was to use a large processor grid. The other was to employ the CT control strategy, since fewer reincarnations result in fewer incorrect (e.g., too low) confidence levels.

Fixes: This attribute is the percentage of correctly-calculated fixes of an emitter.

Fixes can be computed when an emitter has seen at least two observations in the same time interval. If an emitter is undergoing reincarnation, it will not accumulate enough data to regularly compute fixes. Thus, the approaches which minimized reincarnation maximized the correct calculation of fix information.

Fusion: This attribute is the percentage of correct clustering of emitter agents to cluster agents.

The correct computation of fusion appeared to be related, in part, to the correct computation of confidence levels. The fusion process is also the most knowledge-intensive computation in ELINT, and our imperfect results indicate the extent to which ELINT's knowledge is incomplete.

We interpret from Table 6.1 that control strategy has the greatest impact on the quality of results. The CT strategy produced high-quality results irrespective of the number of processors used. The CC strategy, which is much more sensitive to processing delays, performed nearly as well only on the 6×6 processor grid. We believe the added complexity of the CT strategy, while never detrimental, is only beneficial when the interpretation system would otherwise be overloaded by high data rates or poor load balancing.

Tables 6.2 and 6.3 indicate that cost of the added control in the CT strategy is far outweighed by the benefits in its use. Far less message traffic is generated, and the overall simulation time is reduced (In Table 6.2, the last observation is fed into the system at 3.6 seconds; hence, this is the minimum possible simulated run time for the interpretation problem).

Finally, Table 6.4 illustrates the effect of processor grid size when the CT control strategy is employed. This table was produced with the data rate set ten times higher than that used to produce tables 6.1-6.3, the minimum possible simulated run time for the interpretation problem is 0.36 seconds. The speedup achieved by increasing the processor grid size is nearly linear with the square root of the size; however, the 6×6 grid was slightly slower than the 5×5 grid. In this last case, we believe the data rate was not high enough to warrant the additional processors.

6.3 Unanswered Questions

CAOS has been a suitable framework in which to construct concurrent signal interpretation systems, and we expect many of its concepts to be useful in our future computing architectures. Of principal concern to us now is increasing the efficiency with which the underlying CARE architecture is used. In addition, our experience suggests a number of questions to be explored in future research:

- What is the appropriate level of granularity at which to decompose problems for CARE-like architectures?
- What is the most efficient means to synchronize the actions of concurrent problem solvers when necessary?
- How can flexible scheduling policies be implemented without significant loss of efficiency? What is the impact on problem solving if alternate scheduling policies are not provided?

We have started to investigate these questions in the context of a new CARE environment. The primary difference between the original environment and the new environment is that the *process* is no longer the basic unit of computation. While the new CARE system still supports the use of *processes*, it emphasizes the use of *contexts*: computations with less state than those of *processes*.

When a context is forced to suspend to await a value from a stream, it is aborted, and restarted from scratch later when a value is available. This behavior encourages fine-grained decomposition of problems, written in a functional style (individual methods are small, and consist of a binding phase, followed by an evaluation phase).

In addition, CARE now supports arbitrary prioritization of messages delivered to streams. As a result it is no longer necessary to include in CAOS its complex and expensive scheduling strategy. Early indications are that the new CARE environment with a slightly modified CAOS environment performs between two and three orders of magnitude faster than the configuration described in this paper.

Acknowledgements

My thanks to Harold Brown, Bruce Delagi, and Reid Smith for reading and commenting on earlier drafts of this paper. Bruce Delagi, Sayuri Nishimura, Russell Nakano, and James Rice created and maintain the CARE environment. Harold Brown defined the behavior of the CAOS operators, ported ELINT from AGE to CAOS, and collected the results which appear in Section 6.2. Finally, I wish to thank the staff of the Symbolic Systems Resources Group of the Knowledge Systems Lab for their excellent support of our computing environment.

Appendix A

Mergesort: A Simple CAOS Application

Mergesort is an efficient sorting algorithm. It is simple, and well-suited to a concurrent, message-passing implementation. As *mergesort* is not a real-time application, we need not be concerned with the effects of any data rate. Further, its run time is determined entirely by the size of the input; it is not sensitive to initial sorting of the data.

Our algorithm recursively subdivides the input list into two half-size lists, until lists of length 2 are obtained. These lists are then trivially sorted, and recombined in sorted order as the recursion is unwound. We exploit the concurrent CAOS architecture by implementing the recursion as post-value messages sent to other agents. Each processor contains a single *mergesort* agent. Agents are assigned in a globally round-robin order, and are created when necessary by a *mergesort-manager*; we employ one manager per column in the processor grid (this makes use of a natural invariant which lets us replicate managers—see our discussion of this approach within ELINT, in Section 3.2). The algorithm adapts automatically to different processor grids.

Table A.1 illustrates *mergesort*'s runtime on different processor grids and on various input lengths. *mergesort* is well-known to require $O(n \log n)$ time on a uniprocessor; similar analysis indicates *mergesort* should require $O(n)$ time on an "infinite" number of processors.¹ On a grid of size 1, *mergesort* implements a very expensive approach to a conventional *mergesort* (examine the leftmost column of the table); however, on a sufficiently large grid, the algorithm distributes computation across enough processors efficiently enough to achieve nearly $O(n)$ time (as seen in diagonal boundary of the table).

Table A.1 also illustrates the effects of choosing too small a grain-size for CAOS. *mergesort* is dominated by both communication and agent creation costs. It took substantially longer to sort an 8-element list on 4 processors than on 1 processor. Most of this time was spent waiting for answers from *mergesort-manager* agents.

¹ An infinite number of processors is a sufficient number to prevent any runnable "process" from having to wait for a free processor; in our implementation of *mergesort*, this number is $n/2$. Shapiro's implementation in Concurrent Prolog achieved $O(n)$ time with $O(\log n)$ processors [12].

n	Processor Grid Size					
	1	4	9	16	25	36
64	1414	912	756	640	537	514
32	803	606	466	432	471	
16	460	388	349	344		
8	274	397	242			
4	121	141				
2	31					

Table A.1: `mergesort` runtimes (in milliseconds) on various processor grids and input sizes.

A.1 The mergesort Source Code

This section contains the source code for `mergesort`. It is intended to show the flavor of programming in CAOS with a relatively simple example. We show first the code which declares and executes within the `mergesort` and `mergesort-manager` agents.

```

;;; Global variables controlling assignment of agents to sites
;;;
;;; If we were strict, this wouldn't be possible, since we're
;;; making use of the fact that memos in each site really isn't
;;; distributed. However, we do this to force round-robin
;;; allocation.
(defconst *last-x* 1)
(defconst *last-y* 1)
(defconst *array-width* 1)
(defconst *array-height* 1)

;;; Define the basic mergesort agent
(defagent mergesorter (vanilla-agent)
  (documentation "An agent which can perform a level of mergesorting")
  (symbolically-referenced-agents
    ((mergesorter-1-1) mergesorter)
    ((mergesort-manager-1) mergesort-manager)
    ((mergesort-manager-2) mergesort-manager)
    ((mergesort-manager-3) mergesort-manager)
    ((mergesort-manager-4) mergesort-manager)
    ((mergesort-manager-5) mergesort-manager)
    ((mergesort-manager-6) mergesort-manager))
  (instance-vars
    (known-sorters vp-slot value nil datatype #dictionary)
    (managers vp-slot value '((1 . mergesort-manager-1)
                               (2 . mergesort-manager-2)
                               (3 . mergesort-manager-3)
                               (4 . mergesort-manager-4)
                               (5 . mergesort-manager-5)
                               (6 . mergesort-manager-6))
              datatype #dictionary))
  (messages-methods (mergesort :mergesort)))

```

```

;;; The initialize method clears the dictionary of site-agent
;;; mappings prior to the start of each run.
(defmethod (mergesorter :initialize) (&rest ignore)
  (send self 'known-sorters :initialize))

;;; The next-neighbor method returns a stream to a sorting agent
;;; which will perform half of the next lower-level recursive sort.
(defmethod (mergesorter :next-neighbor) ()
  (let ((next-location-site
        (multiple-value-bind (x y) (next-x-and-y)
          ;; x and y hold site coordinates for the next agent.
          (send (lookup-site x y) :care-site))))
    (let ((maybe-known-agent
          ;; check the dictionary for a site-agent mapping.
          (send self 'known-sorters :get next-location-site)))
      (cond (maybe-known-agent maybe-known-agent)
            (t (let ((next-location
                      (send next-location-site :location)))
                  ;; Don't know the mapping. Ask a manager.
                  (send self 'known-sorters :put
                           next-location-site
                           (post-value (send self 'managers :get
                                                (first next-location))
                                         nil
                                         :new-agent (first next-location)
                                         (second next-location))))))))))

```

```

(defmethod (mergesorter :mergesort) (&rest list)
  (cond ((eq (length list) 2)
    ;; Trivial case. Lists of length 2.
    '(. (min (first list) (second list))
      . (max (first list) (second list))))
    (t (let* ((first-neighbor (send self :next-neighbor))
              (second-neighbor (send self :next-neighbor)))
      ;; Recurse: divide the list and sort both halves.
      ;; Use post-future to start each half.
      (first-future
        (lexpr-funcall #'post-future first-neighbor nil
          :mergesort
          (copylist (first-half list))))
      (second-future
        (lexpr-funcall #'post-future second-neighbor nil
          :mergesort
          (copylist (second-half list)))))
      ;; Combine the sorted sublists.
      ;; value-future blocks until the half finishes.
      (do ((e1 (value-future first-future))
          (cond ((null e2) (cdr e1))
                ((or (null e1) (> (first e1) (first e2)))
                 e1)
                (t (cdr e1))))
          (e2 (value-future second-future))
          (cond ((null e1) (cdr e2))
                ((or (null e2) (> (first e2) (first e1)))
                 e2)
                (t (cdr e2))))
          (result nil))
          ((and (null e1) (null e2)) result)
          (cond ((and e1 e2)
            (setq result (nconc result
              (list (min (first e1)
                (first e2))))))
            (e1 (setq result (nconc result
              (list (first e1))))))
            (t (setq result (nconc result
              (list (first e2))))))))))

;;; Function to maintain globally round-robin agent-site
;;; allocation.
(defun next-x-and-y ()

```

```

(multiple-value-prog1 (values *last-x* *last-y*)
  (when (> (incf *last-x*) *array-width*)
    (setq *last-x* 1)
    (when (> (incf *last-y*) *array-height*)
      (setq *last-y* 1)))))

;;; Return the first half of a list.
(defun first-half (list)
  (loop for i from 1 to (// (length list) 2) as e in list
    collect e))

;;; Return the second half of a list.
(defun second-half (list) (nthcdr (// (length list) 2) list))

;;; Define the mergesort-manager. These agents, located one
;;; per column in the processor grid, are responsible for
;;; creating new mergesort agents upon request.
(defagent mergesort-manager (vanilla-agent)
  (documentation "An agent to create other mergesorters")
  (instance-vars agent-array)
  (messages-methods (new-agent :new-agent)))

;;; The initialize method clears the manager's mapping of
;;; (x,y) coordinates to mergesort agent.
(defmethod (mergesort-manager :initialize) (max-x max-y)
  (setq agent-array (make-array (list (1+ max-x) (1+ max-y)))))

;;; The new-agent method returns the agent already at
;;; (x,y), or creates a new agent at (x,y) and returns it.
(defmethod (mergesort-manager :new-agent) (x y)
  (cond ((aref agent-array x y))
        (t (let ((the-new-agent (create-agent-instance
                                   'mergesorter
                                   (list x y))))
              (aset the-new-agent agent-array x y)
              the-new-agent))))

```


This next section of code is the CAOS initialization file which produced the runtime numbers displayed in Table A.1:

```
(defconst *the-original-list*
  '( 6 7 4 1 2 8 5 3 16 12 9 11 15 13 10 14
    32 22 30 21 28 19 26 18 24 31 22 29 20 29 25 17
    64 63 62 61 60 59 34 33 58 57 56 55 54 53 52 51
    50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35))

(defconst *the-current-list* nil)

(caos-initialize
  ((mergesorter-1-1 mergesorter (1 1))
   (mergesort-manager-1 mergesort-manager (1 1))
   (mergesort-manager-2 mergesort-manager (2 1))
   (mergesort-manager-3 mergesort-manager (3 1))
   (mergesort-manager-4 mergesort-manager (4 1))
   (mergesort-manager-5 mergesort-manager (5 1))
   (mergesort-manager-6 mergesort-manager (6 1)))
  ((with-open-file (log "x7:schoen.qsort;qsort.log" :write)
    (setq *the-current-list* *the-original-list*)
    (loop with start-time for j from 6 downto 1 do
      (format log "~&Sorting the list:~&S"
        *the-current-list*)
      (loop for i from 1 to j do
        (multipost-value
          '(mergesort-manager-1 mergesort-manager-2
            mergesort-manager-3 mergesort-manager-4
            mergesort-manager-5 mergesort-manager-6)
          nil :initialize i i)
        (post-value mergesorter-1-1 nil :initialize)
        (format log "~&Starting ~D processor sort at ~D"
          (* i i) (caos-time))
        (setq start-time (caos-time))
        (lexpr-funcall #'post-value mergesorter-1-1 nil
          :mergesort *the-current-list*)
        (format log "~&Finished at ~D. That took ~D ms"
          (caos-time)
          (* (- (caos-time) start-time) 1.0e-5)))
      (setq *the-current-list* (first-half *the-current-list*))))))
```

We conclude with the log file produced by this mergesort execution:

Sorting the list:

(6 7 4 1 2 8 5 3 16 12 9 11 15 13 10 14 32 22 30 21 28 19 26 18 24 31
22 29 20 29 25 17 64 63 62 61 60 59 34 33 58 57 56 55 54 53 52 51 50
49 48 47 46 45 44 43 42 41 40 39 38 37 36 35)

Starting 1 processor sort at 9803527

Finished 1 processor sort at 151163188. That took 1413.5966 ms

Starting 4 processor sort at 157430828

Finished 4 processor sort at 248600531. That took 911.697 ms

Starting 9 processor sort at 254848384

Finished 9 processor sort at 330631571. That took 757.83185 ms

Starting 16 processor sort at 337017977

Finished 16 processor sort at 401035492. That took 640.1752 ms

Starting 25 processor sort at 407972369

Finished 25 processor sort at 461663705. That took 536.9133 ms

Starting 36 processor sort at 468137724

Finished 36 processor sort at 519548649. That took 514.10925 ms

Sorting the list:

(6 7 4 1 2 8 5 3 16 12 9 11 15 13 10 14 32 22 30 21 28 19 26 18 24 31
22 29 20 29 25 17)

Starting 1 processor sort at 526138721

Finished 1 processor sort at 606424159. That took 802.3544 ms

Starting 4 processor sort at 613038165

Finished 4 processor sort at 673645208. That took 606.07043 ms

Starting 9 processor sort at 680223869

Finished 9 processor sort at 726796432. That took 465.72562 ms

Starting 16 processor sort at 733697221

Finished 16 processor sort at 776848166. That took 431.50943 ms

Starting 25 processor sort at 783605583

Finished 25 processor sort at 830669664. That took 470.64078 ms

Sorting the list:

(6 7 4 1 2 8 5 3 16 12 9 11 15 13 10 14)

Starting 1 processor sort at 837629049

Finished 1 processor sort at 883646903. That took 460.17856 ms

Starting 4 processor sort at 890496880

Finished 4 processor sort at 929338867. That took 388.41986 ms

Starting 9 processor sort at 936242285

Finished 9 processor sort at 971092553. That took 348.5027 ms

Starting 16 processor sort at 978109126

Finished 16 processor sort at 1012524715. That took 344.15538 ms

Sorting the list:

(6 7 4 1 2 8 5 3)

Starting 1 processor sort at 1019622193

Finished 1 processor sort at 1046974695. That took 273.52502 ms
Starting 4 processor sort at 1054797480
Finished 4 processor sort at 1094519241. That took 397.2176 ms
Starting 9 processor sort at 1101582612
Finished 9 processor sort at 1125786372. That took 242.0376 ms
Sorting the list:
(6 7 4 1)
Starting 1 processor sort at 1132929674
Finished 1 processor sort at 1145004341. That took 120.746666 ms
Starting 4 processor sort at 1152132853
Finished 4 processor sort at 1166264559. That took 141.31706 ms
Sorting the list:
(6 7)
Starting 1 processor sort at 1173565420
Finished 1 processor sort at 1176647734. That took 30.82314 ms

Appendix B

Implementing the CAOS Framework

This appendix is a guide to the source files which implement the CAOS system. The descriptions which follow are at a much greater level of detail than those in Chapter 5, and are intended primarily for readers of the source code, as a supplement to the embedded documentation. It is assumed that readers of this appendix have a familiarity with Lisp (principally ZETALISP or CommonLisp), and have read Chapter 5.

B.1 General Programming Issues

All data structures are implemented with the `defstruct` mechanism. `defstruct` accepts a description of the desired data structure, and produces a number of macro definitions which serve to create new instances of the structure, and access and modify fields of the structure. For example, a `ship` data structure may be defined as having fields `name`, `position`, and `course`. New instances of `ship`'s are created by calling `make-ship`; the fields of the ship structure are accessed by calling `ship-name`, `ship-position`, and `ship-course`. A field may be modified by embedding a field access function in a `setf` expression.

The CAOS system is intended for use in ZETALISP-compatible environments. The system was developed originally on the Symbolics 3600 family of workstations, and was later ported to the Texas Instruments *Explorer* workstation. These machines each support a ZETALISP programming environment, but are not completely source-code compatible.

Source-level incompatibilities are handled by use of the `#+` and `#-` reader macros. An occurrence of `#+Symbolics` in a source file causes the next `s-expression` to be read only when the file is being loaded into a Symbolics workstation, an occurrence of `#-Symbolics` prevents the following `s-expression` from being loaded into a Symbolics workstation. Similar read-time conditionals for the TI environment are introduced by `#+TI` and `#-TI` constructs.

B.2 Interface to CARE

In order to function properly under the CARE simulator, all CAOS code and CAOS applications must be loaded into the `care-user` symbol package. This package is defined to inherit from CARE those symbols (e.g., functions, variables, and macros) which comprise the exported CARE programming interface.

B.2.1 CARE Data Structures

The following CARE-defined data structures are used CAOS:

`remote-address` [Structure]

A `remote-address` is the global encapsulation for the address of a data structure located on a particular processor. It may be thought of as extending the address space of a site with additional address bits that identify the site in the processor grid.

`remote-address`'s contain two fields: `site` and `local`. The `site` field identifies the site on which the structure pointed to by the `local` field resides.

`site` [Structure]

A `site` represents one of the processing nodes in the grid. An instance of a `site` structure is actually an instance of a `site` flavor, and hence, fields of a `site` are accessed by sending Flavors messages. The following are messages relevant to CAOS: `:location`, which returns the (x, y) coordinate of the site in the grid; `:x-site`, which returns the x coordinate of the site; and `:y-site`, which returns the y coordinate.

`queue` [Structure]

A `queue` implements FIFO storage, and is used in a number of places within CARE. In particular, packets arriving on a CARE stream are stored in a queue. The `queue` structure has the following relevant fields: `length`, `body`, `tail`. The `length` field stores the number of entries which are currently in the queue; the `body` field points to a list which implements storage for the queue, the `tail` field points to the last element of the body of the queue, and allows new entries to be appended to the end of queue in $O(1)$ time (Access to the head of the queue also requires $O(1)$ time).

`stream` [Structure]

A `stream` is a virtual circuit which carries data (in the form of packets) between processes. Operations on streams are performed by the functions `post-packet` and `accept-packet`, which are described below. The `packets` field of a stream contains the queue of packets which have arrived on the stream. The `properties` field of a stream contains an arbitrary property list; CAOS uses the property list to store information to help the function which prints out streams in a human-readable fashion. Other fields of the stream are not relevant to CAOS.

process

[Structure]

A **process** is the basic unit of computation in CARE. The innards of a **process** are of no concern to CAOS; however, it should be noted that the special variable *****care-process***** is always bound to the **process** structure of the **process** currently executing.

B.2.2 CARE Functions and Macros

The following functions and macros are used by CAOS:

post-packet *&optional form &key ...*

[Macro]

The macro **post-packet** is used to create new streams and new processes, and to exchange messages between processes. If called with no arguments, it returns a new **stream** instance. All other **post-packet** options are controlled by the existence of various keywords in its argument list. When keyword arguments are supplied, the first argument to **post-packet** is evaluated to form the message to be sent.

The following keyword options are employed by CAOS:

to: The value of the **to** keyword is a stream or list of streams to which the message will be sent.

for: The value of the **for** keyword is a stream or list of streams. When the message is received remotely, the value of this keyword will appear in the **clients** field of the message.

for-new-stream, **process**: These two keywords always appear together in an argument list, and take no arguments. They are included in a call to **post-packet** to create new processes. The first argument in such a call is a form to evaluate remotely to start the process. This call also requires a **to** keyword argument, which must be a **remote-address**; the process is created on the site indicated by the **site** field.

The value of the call is a **stream**. A call to **accept-packet** on this stream will return a packet whose **value** field is the default stream supplied to the newly-created process.

after: The value of the **after** keyword is a time interval, in microseconds. When this keyword is supplied, the message will be delivered after a corresponding delay. The purpose of the keyword is to provide for a means of implementing *timeouts*. A process can cause a packet to be posted to a stream only after a specified interval; when this packet arrives, any processes waiting on the stream will be awakened. CAOS implements "clocked futures" using this mechanism.

tagged: The **tagged** keyword provides a means of tagging the message with a user-supplied value; its principal use is in debugging and message tracing.

with-packet-bindings *stream-form bindings &body forms*

[Macro]

The `with-packet-bindings` macro evaluates *stream-form*, which must return a *stream*. It then picks the first packet from the stream (or blocks the calling process until a packet arrives), and (lambda) binds portions of the packet to the variables specified in *bindings*. The format of *bindings* is a list. The first variable name in the list is bound to the contents of the message; the second is bound to the clients of the message (e.g., the streams specified by the `for` keyword in the call to `post-packet`). Additional variables may be bound to fields which are not relevant in the discussion of CAOS.

`accept-packet stream` [Function]

The macro `with-packet-bindings` is defined in terms of this function. `accept-packet` is called with *stream* bound to a *stream*, and returns the first packet waiting in the stream (or blocks the calling process until a packet is available).

`defprocess` [Macro]

The `defprocess` macro is syntactic sugar for `defun`. Any function which is to be the top-level of a CARE-process should be defined using `defprocess`. The last argument in the argument list of a function defined by `defprocess` will be bound to the default stream for the process; thus, any function defined with `defprocess` must have at least one argument.

B.3 The CAOS Support Environment

In Chapter 5, we described an extension to Flavors which implements abstract data type support for instance variables. The files `herbs.lisp`, `sage.lisp`, `datatype.lisp`, and `priority-queue.lisp` comprise the framework which includes abstract data type support. In addition, these files contain code which implements a sort of inheritance of default values of instance variables, and code which implements substructure for instance variables.

B.3.1 Herbs.Lisp

This file implements a form of inheritance of list-structured default values of instance variables. The Flavors class hierarchy forms a taxonomy; classes defined far from the root of the taxonomy are more specialized than those defined near the root. Within a class, methods can be combined with methods of the same name in ancestral classes in quite a few ways. Unfortunately, Flavors provides no means of combining inherited values.

Consider the example of Figure B.1. The Flavor class `flavor-3` is defined as a subclass of classes `flavor-1` and `flavor-2`. Both `flavor-1` and `flavor-2` define an instance variable called `iv-a`. What value does `flavor-3` inherit as the default for `iv-a`?

In normal Flavors, `flavor-3` would inherit `'(a b c)` as the default value. However, there are situations in which the proper value to inherit for `iv-a` might be `'(a b c d e f)`. The `defherb` macro, defined in `herbs.lisp`, enables this sort of inheritance.

Figure B.2 illustrates three possible inheritance modes for the default value of `iv-a` in `flavor-3`. In the first example, the default value of `iv-a` will be `'(a b c d e f)`. In the second example, its value will be `'(a b c d e f g h i)`. In the final example, its value will be `'(b d f)`.

```
(defflavor flavor-1 ((iv-a '(a b c))) ())

(defflavor flavor-2 ((iv-a '(d e f))) ())

(defflavor flavor-3 () (flavor-1 flavor-2))
```

Figure B.1: Multiple inheritance example.

```
(defherb flavor-3 ((iv-a + nil)) ())

(defherb flavor-3 ((iv-a + '(g h i))) ())

(defherb flavor-3 ((iv-a - '(a c e))) ())
```

Figure B.2: defherb examples.

B.3.2 Sage.Lisp

This file implements structured and abstract data type support for instance variables. Both facilities depend on storing special-purpose structures, known as **vp-slot**'s, in instance variables. Descriptions of the **vp-slot** structure, and the important functions which access it, follow (many of the concepts used here come from the Strobe system [13]):

vp-slot [Structure]

A **vp-slot** contains three primary fields. The **value** field holds the "value" of the slot. The **datatype** field holds an indication of what sort of objects will reside in the **value** field of the slot. Finally, the **user-defined-facets** field holds an association list of arbitrary facet names and values; new facets may be added at any time.

A **vp-slot** may be thought of as a value with arbitrary annotations (All slots are annotated with a **datatype** facet). These annotations might permit a program to reason about the contents of the slot when necessary.

getfacet *object slot &optional (facet 'value) errorflag novalueflg* [Function]

The function **getfacet** returns the value of *facet* in *slot* of *object*. *Facet* defaults to **value**, which retrieves the **value** field of the **vp-slot**. Other acceptable bindings for *facet* are **datatype**, plus any facet in the **user-defined-facets** field of the slot. If the facet doesn't exist, and the value of *errorflag* is non-**nil**, a fatal error will occur. If the value of the facet is ***novalue***, and *novalueflg* is **nil**, the value returned from **getfacet** will be **nil**; otherwise, it will be the value found in the facet.

putfacet *object slot &optional (facet 'value) (value '*novalue*') errorflag* [Function]

The function `putfacet` puts *value* into *facet* of *slot* of *object*. If the facet doesn't exist, it is first created. If the slot doesn't exist (e.g., the instance variable named *slot* doesn't exist, or doesn't contain an object of type `vp-slot`) and *errorflag* is non-nil, a fatal error is signalled.

#_

[Reader Macro]

Unfortunately, by placing `vp-slot` structures in instance variables of Flavor instances, it becomes impossible to simply get the "value" of the instance variable (since the value is now a `vp-slot`). The `#_` reader macro is a piece of syntactic sugar which expands to the form `(vp-slot-value ...)`, and hence, retrieves the *value* field of the slot. Therefore, references to instance variables which contain slots can be preceded by `#_` to retrieve the actual value of the slot.

A number of macros are defined in terms of these basic functions; their function should be clear from examination of the source code.

Abstract Data Type Support

Abstract data type support for instance variables is implemented by forwarding messages sent to `vp-slot`'s to the objects pointed to by their *datatype* fields. Consider the example in Figure B.3. The inclusion of the `:gettable-instance-variables` option in the definition of `flavor-1` causes instances of `flavor-1` to respond to `:iv-a` messages (note the ':' in the message name); instances of `flavor-1` do not respond to the `iv-a` message.

Normally, when a message for which no method is defined is sent, an error occurs; however, it is possible to define an `:unclaimed-method` method for a Flavors class. The `:unclaimed-method` is invoked when an undefined message is sent. The file `sage.lisp` defines a Flavors class, `sage-class`, which has just this sort of `:unclaimed-method`.

When an undefined message is sent to a Flavors instance which has `sage-class` as an ancestor, the following steps are taken:

1. If the message is actually the name of an instance variable in the instance, the message name is evaluated (using `symbol-in-instance`) to retrieve the value of the variable.
2. If the value of the variable is a structure of type `vp-slot`, a message is sent to the Flavors instance stored in the *datatype* field of the slot. The message name is taken from the first "argument" of the unclaimed message. The arguments in the message are the Flavors instance to which the message was originally sent, the name of the instance variable to which the message was sent, and all but the first of the original arguments of the unclaimed message.

Now consider the course of events when `(send instance-1 'iv-a :get 'b)` is evaluated:

1. The message `iv-a` is received by `instance-1`.
2. `instance-1` does not handle the message `iv-a`, so the message is forwarded to the `:unclaimed-method` method defined by `sage-class`.

```

(defflavor association-list () ())

(defmethod (association-list :get) (instance iv key)
  (cdr (assq key (getvalue instance iv))))

(defvar assn-instance (make-instance 'association-list))

(defflavor flavor-1
  ((iv-a (make-vp-slot value '((a . 1) (b . 2) (c . 3))
                        datatype assn-instance)))
  (sage-class)
  :gettable-instance-variables)

(defvar instance-1 (make-instance 'flavor-1))

```

Figure B.3: A Flavor containing a slot

3. The `:unclaimed-method` code evaluates `iv-a` in the context of `instance-1`, and discovers the value to be a structure of type `vp-slot`. It then effectively evaluates the following: `(send assn-instance :get instance-1 'iv-a 'b)`.
4. The `:get` method of `association-list` is called. It uses its first two arguments to retrieve the association list from the value field of the `vp-slot` to which the message was originally directed. It then uses its third argument to return the value of an association from the list.
5. The value returned by the `:get` method of the `vp-slot`'s datatype is returned as the value of the original message.

A number of macros are defined for the convenience of programmers:

`defdatatype`

[Macro]

Defines a new Flavors class suitable for use as an abstract data type. This is syntactic sugar for a combining `defflavor` and `defmethod` into one textual unit. For example, the above definition of `association-list` could have been made by evaluating:

```

(defdatatype association-list "Implements a-list dictionaries."
  (:get (instance iv key)
    (cdr (assq key (getvalue instance iv)))))

```

##

[Reader Macro]

This reader macro accepts the name of a datatype class, and returns an instance of the class. If no instances of the class have been created, it creates one and stores it in a hash table (**sage-datatype-hash-table**). This reader macro is used in creating slots:

```
(deflavor flavor-1
  ((iv-a (make-vp-slot value '((a . 1) (b . 2) (c . 3))
                        datatype #association-list)))
  ()))
```

B.3.3 Datatype.Lisp and Priority-Queue.Lisp

These files use the facilities defined by *sage.lisp* and *herbs.lisp* to define a number of useful abstract data types. In general, these ADT's respond to an *:initialize* message to initialize themselves to an "empty" state, a *:put* message to add items to themselves, and a *:get* message to remove items from themselves.

queue

[Abstract Data Type]

The queue data type implements FIFO storage in an instance variable. The current implementation uses lists maintained by the *tconc* function, defined in *datatype.lisp*. The *:initialize* message empties the queue, the *:put* message enqueues entry on the end of the queue, and the *:get* message dequeues an entry from the front of the queue. If the instance variable in which the queue resides has a *max-length* facet, entries are added to the queue if-and-only-if the current length of the queue is less than the specified maximum length.

Two values are returned by a *:put* message. The first value is *t* if there was room to append the new entry; the second value is the value appended to the queue. Two values are also returned by the *:get* message. The first is the value found at the head of the queue; the second is *nil* if the queue was empty before the message, or *t* if it was non-empty.

All operations defined for a queue require $O(1)$ time.

dictionary

[Abstract Data Type]

The dictionary is a fuller version of the *association-list* ADT described above. The *:put* and *:get* operations require $O(n)$ time, and hence, suggest the dictionary datatype be used when the number of entries is expected to be small. In addition to *:initialize*, *:put*, and *:get* messages, the dictionary also responds to the following messages:

:add key value

[Datatype Message]

Adds *value* as an additional value to be associated with *key*. A *:get* message on *key* will subsequently return lists of two or more values. Requires $O(n)$ time.

:forget key

[Datatype Message]

Removes the entry associated with *key* from the dictionary. Requires $O(n)$ time.

:map function [Datatype Message]

Applies *function* to each entry in the dictionary. *Function* must be a function of two arguments; the first argument will receive the key of an entry, and the second will receive the value of the key. Requires $O(n)$ time.

:new-id [Datatype Message]

Returns a key which is guaranteed not to be in the dictionary. This is currently implemented using *gensym*, and as such, requires $O(1)$ time.

:number-of-entries [Datatype Message]

Returns the number of entries in the dictionary. Requires $O(1)$ time.

:all-entries [Datatype Message]

Returns all of the entries in the dictionary, in association-list format. Requires $O(1)$ time.

sorted-dictionary [Abstract Data Type]

The **sorted-dictionary** is a variant of the dictionary which keeps its entries in sorted order, as defined by a user-supplied comparison function. It responds to the same messages as does the dictionary. The time complexity of operations defined for a **sorted-dictionary** are equivalent to those defined for a dictionary.

The comparison function must be a predicate of two arguments, and must return *t* if-and-only-if the first argument is "greater" than the second argument. For example, if the keys represent timestamps, and the dictionary is to keep the keys sorted in ascending order, the comparison function can be specified as *#'<*, the *lessp* function.

In addition to the messages defined by the dictionary data type, the **sorted-dictionary** also responds to these messages:

:greatest-entry [Datatype Message]

The **:greatest-entry** message returns the key having the "greatest" value, as defined by the comparison function. Because the dictionary is kept in sorted order, this operation requires only $O(1)$ time.

:next-entry n [Datatype Message]

The **:next-entry** message returns the key of the entry having the next "greatest" value to that of *n*. This is an $O(n)$ operation.

hash-dictionary [Abstract Data Type]

The hash-dictionary is a dictionary implementation which is based on hash tables, rather than association lists. It responds to the same messages as does the dictionary ADT. Its advantage over the dictionary is that insertion, lookup, and deletion operations are all of $O(1)$ time complexity; however, the enumeration message, `:all-entries`, is of $O(n)$ time complexity.

monitor

[Abstract Data Type]

The **monitor** data type is a special purpose ADT which aids in the implementation of lexically-scoped mutual exclusion. Storage for the monitor is implemented by a **monitor** structure:

monitor

[Structure]

The monitor structure contains two fields: **owner**, which points to the process which currently owns the monitor; and **waiting-processes**, which is a queue of processes waiting to obtain ownership of the monitor.

:enter wakeup-stream

[Datatype Message]

A process wishing to enter a region of mutual exclusion sends this message. If the monitor is unowned, the owner is set to the value of `***care-process***`, and the caller is allowed to enter the region of mutual exclusion.

If the monitor is currently owned, a dotted pair, consisting of the value of `***care-process***` and `wakeup-stream`, is enqueued on the **waiting-processes** queue of the monitor. The caller then calls `accept-packet` in order to suspend execution. When the caller's request reached the head of the queue, a packet will be sent to `wakeup-stream`, restarting the suspended caller.

:exit

[Datatype Message]

The `:exit` message relinquishes ownership of the monitor, and restarts the next process waiting to obtain it (if any).

If the **waiting-processes** queue is non-empty, the first entry on the queue is dequeued. The entry contains the process handle of the waiting process, which is placed in the **owner** field of the monitor, and the **stream** upon which to send the wakeup message.

If the queue is empty, the **owner** field of the monitor is set to nil, so that the monitor is marked as unowned.

with-monitor monitor-name &body forms

[Macro]

This macro implements an error-protected, lexically-scoped mutual exclusion. *Monitor-name* must be the name of an instance variable in the Flavors instance currently bound to `self` which holds a **monitor**. Upon entry to this macro, an `:enter` message is sent to the monitor to gain entrance. The expressions in *forms* are then executed under `unwind-protect` protection, such that if an error occurs during their execution, the monitor is guaranteed to be released.

This macro is equivalent to the `with.monitor` macro of Interlisp-D.

without-monitor *monitor-name* &body *forms*

[Macro]

This macro is intended to be used within the scope of a **with-monitor** form. Its purpose is to temporarily release ownership of the monitor specified by *monitor-name* (using the **:exit** method), and then to reobtain it (using the **:enter** method) after the forms in *forms* have been executed. Typically, *forms* will contain an expression that causes the calling process to suspend for some period of time (or until a packet arrives on some stream).

This macro is similar in spirit to the **monitor.await.event** macro of Interlisp-D.

priority-queue

[Abstract Data Type]

The **priority-queue** data type and the code needed to implement it are contained on the file **priority-queue.lisp**. The build of this file is a set of ZetaLisp routines which implement a dynamic, *Heapsort*-style priority queue. The implementation is derived from algorithms **DELETMIN** and **INSERT**, from section 4.11 of [1]. Insertion and deletion from this queue both require $O(n \log n)$ time.

priority-queue

[Structure]

The **priority-queue** structure implements storage at the nodes of the partially-ordered binary tree. It has fields **left-child**, **right-child**, and **item**. In addition, for convenience, it has a **priority-function** field which stores the priority-computing function for entries in the tree.

exchange-nodes *top bottom*

[Macro]

This macro exchanges the contents of nodes *top* and *bottom*.

insert-in-queue *queue node*

[Function]

This function inserts *node*, an instance of a **priority-queue** structure, into the tree rooted by *queue*. It recursively descends into the tree, heading for the leftmost free node at the lowest level of the tree (creating a new level if necessary). As it unwinds from the recursion, it exchanges nodes as necessary to maintain the partial order. The value returned from this function is the new root of the tree, which may have changed.

rebalance-queue *queue*

[Function]

This function rebalances the tree rooted at *queue* after its root has been removed.

remove-from-queue *queue*

[Function]

This function removes the item from the partially-ordered tree rooted at *queue*, and rebalances the tree to maintain the partially-ordered invariant. It returns two values, the value found at the root, and a pointer to the new root of the tree.

```

sorting-spec ::= (key-spec . sorting-spec) | nil
key-spec ::= (key-name . field-spec-list)
field-spec-list ::= (field-spec . field-spec-list) | nil
field-spec ::= (field-computation . predicate)
field-computation ::= field-arg | (field-op . field-arg-list)
field-arg-list ::= (field-arg . field-arg-list) | nil
field-op ::= any-lisp-function
key-name ::= any-lisp-symbol
field-arg ::= field-number | 'any-valued-lisp-symbol
field-number ::= any-lisp-integer
predicate ::= any-lisp-predicate

```

Figure B.4: BNF Grammar for declaring sorting functions.

```

((:site ((+ (* 0 '16) 1) . <))
 (:agent (2 . alphalessp))
 (:task (3 . <)))

```

Figure B.5: A sample sorting specification.

B.4 Instrumentation for CAOS

The CARE system comes supplied with a wide variety of "instrument panels" which report how various components of the simulated execution architecture are being utilized. Much of CAOS was defined prior to the existence of these instruments, and the file `pravda.lisp` contains vestigial remnants of an interim CAOS-based instrumentation package. This package is no longer in use, and it will not be documented here, although it is part of the CAOS sources. There are, however, CAOS-specific instrument panels which are still in use. These panels are documented in this section.

B.4.1 Scrolling-Text-Panel.Lisp

The file `scrolling-text-panel.lisp` contains an instrument which displays information in a sorted order in a ZETALISP-defined window known as a `tv:scroll-window`. Such windows are designed to display a structured representation of data; new lines of information may be added or deleted dynamically, and the window may be scrolled vertically if more information is being displayed than can fit in the window.

The `scrolling-text-panel` is a `tv:scroll-window` whose sorting order and display formatting commands are specified by a simple, declarative grammar. The declaration of the sorting function is specified in the `:sort` -by instance variable of the panel; the formatting function is specified by the `:printed-by` and `:formatted-by` instance variables. We first describe the grammar as it pertains to sorting.

The sorting grammar is described in BNF format in Figure B.4;¹ an example from CAOS appears in Figure B.5. Unquoted numbers used in *field-number* positions refer to corresponding elements of a vector in which information which drives the sorting and display functions resides.

The sorting declaration in Figure B.5 constructs three sorting functions, indexed respectively by the keywords `:site`, `:agent`, and `:task`. The `:site` sorting function is compiled into the following pieces of Lisp code:²

```
(defun foo-site-sorter (item-1 item-2)
  (let ((entry-1 (array-leader item-1 (1+ tv:scroll-item-leader-offset)))
        (entry-2 (array-leader item-2 (1+ tv:scroll-item-leader-offset))))
    (< (+ (* (nth 0 entry-1) 16) (nth 1 entry-1))
        (+ (* (nth 0 entry-2) 16) (nth 1 entry-2)))))
```

The `:agent` sorting function is a refined version of the `:site` sorting function. It expands into:

```
(defun foo-agent-sorter (item-1 item-2)
  (let ((entry-1 (array-leader item-1 (1+ tv:scroll-item-leader-offset)))
        (entry-2 (array-leader item-2 (1+ tv:scroll-item-leader-offset)))
        (key-2 (array-leader item-2 tv:scroll-item-leader-offset)))
    (cond ((foo-site-sorter item-1 item-2) t)
          ((equal item-1 item-2)
           (cond ((memq key-2 '(:site)) nil)
                 (t (alphalessp (nth 2 entry-1) (nth 2 entry-2)))))))
```

The `:task` sorting function is further refined, and expands to:

```
(defun foo-task-sorter (item-1 item-2)
  (let ((entry-1 (array-leader item-1 (1+ tv:scroll-item-leader-offset)))
        (entry-2 (array-leader item-2 (1+ tv:scroll-item-leader-offset)))
        (key-2 (array-leader item-2 tv:scroll-item-leader-offset)))
    (cond ((foo-agent-sorter item-1 item-2) t)
          ((equal item-1 item-2)
           (cond ((memq key-2 '(:site :agent)) nil)
                 (t (< (nth 3 entry-1) (nth 3 entry-2)))))))
```

We now discuss the language with which formatting functions are defined. Lines of text are output to scrolling-text-panels with the function `format`; in order to use this function, we must have a way of choosing both format control strings and the expressions which are evaluated to generate arguments for these control strings.

¹In this figure, and in Figure B.6, tokens in this font are non-terminals, and tokens in this font are terminals. Occurrences of "." are Lisp "consing dots," thus, where the grammar would ordinarily demand statements of the form (a . (b . (c . nil))), it is acceptable to supply the form (a b c).

²The arguments `item-1` and `item-2` are bound to instances of `tv:scroll-line-item` structures. The internal representation of these structures is unimportant, except that arbitrary application-program information may be stored in their *array leader* sections. The first word of available storage in the array leader is found at `tv:scroll-item-leader-offset`.


```

print-spec ::= (key-spec . print-spec) | nil
key-spec ::= (key-name . field-spec-list)
field-spec-list ::= (field-computation . field-spec-list) | nil
field-computation ::= field-arg | (field-op . field-arg-list)
field-arg-list ::= (field-arg . field-arg-list) | nil
field-op ::= any-lisp-function
key-name ::= any-lisp-symbol
field-arg ::= field-number | 'any-valued-lisp-symbol
field-number ::= any-lisp-integer

```

Figure B.6: BNF Grammar for declaring printing functions.

```

((:site . "SITE-~D-~D")
 (:agent . "      ~A ~A (~D run, ~D wait)")
 (:task . "      ~A ~A ~A"))

((:site 0 1)
 (:agent 2 (car 3) 4 5)
 (:task 4 3 5))

```

Figure B.7: A sample formatting specification

Format control strings are chosen by indexing into an association list stored in the formatted-by instance variable of the panel. Lisp expressions which generate the arguments for `format` are created by parsing expressions defined by the grammar in Figure B.6 and are found in the printed-by instance variable of the panel. The contents of these two instance variables, in an example from the CAOS instrumentation, is illustrated by Figure B.7. The panel defined by the specifications in Figures B.5 and B.7 will display sites in column-major order; within each site, agents will be displayed alphabetized by name; within each agent, tasks will be displayed ordered by arrival time. For example:

```

SITE-1-1
  MERGESORT-MANAGER-1 INITIALIZED (0 run, 0 wait)
  MERGESORTER-1-1 INITIALIZED (1 run, 3 wait)
    RUNNING 345700 NEIGHBOR
    NEVER-RUN 345792 MERGESORT
SITE-1-2
  MERGESORTER-1-2 INITIALIZED (0 run, 0 wait)

```

B.5 CAOS Structures and Macros

The file `czardefs.lisp` contains macro and structure definitions for the rest of the CAOS system.

request-message

[Structure]

The **request-message** structure is a list which defines the contents of messages sent using the various *post* operators of CAOS.

response-message

[Structure]

The **response-message** structure is a list which defines the contents of messages sent as responses to value-desired messages.

caos-time

[Macro]

This macro retrieves the current simulator time, which is measured in simulator clock units. Presently, this figure is measured in 10 nanosecond units.

runnable-item

[Structure]

The **runnable-item** is the CAOS scheduler's handle on a process. Most of its structure was described in Section 5.4. The **panel-entry** field holds the **tv:scroll-window** line entry of the process.

contract

[Resource]

Resources are Lisp objects which must be explicitly allocated and deallocated. This is counter to the normal Lisp philosophy, but is quite useful when the extent of an object is known. The advantage of declaring objects as resources is that large numbers of unused copies of the objects aren't accumulated to be reclaimed only when the garbage collector is run. The **contract** resource allocates and deallocates **runnable-item**'s.

care-site-scrolling-panel-entry

[Structure]

This structure is the vector which holds information for sorting and formatting **care-site** entries in the **scrolling-text-panel**. In figures B.5 and B.7, this structure is referenced by printing and sorting specifications keyed by **:site**. The fields of the structure are:

x, y: Coordinates of the site in the processor grid.

state: The condition of the site.

agent-scrolling-text-panel-entry

[Structure]

This structure is the vector which holds information for sorting and formatting **agent** entries in the **scrolling-text-panel**. It is referenced by printing and sorting specifications keyed by **:agent**. The fields of this structure are:

x,y: Coordinates of the site upon which the agent is located.

name: The name of the agent.

state: The condition of the agent.
nrun: The number of runnable tasks in the agent.
nwait: The number of suspended tasks in the agent.

task-scrolling-panel-entry

[Structure]

This structure is the vector which holds the information for sorting and formatting task (process) entries in the **scrolling-text-panel**. This structure is referenced by printing and sorting specifications keyed by **:task**. The fields of the structure are as follows:

x, y: Coordinates of the site upon which the task is executing.
name: The name of the agent in which the task is executing.
entry-time: The simulator time at which the task started.
state: The current state of the task.
message: The name of the message being executed by the task.

future

[Structure]

A **future** is a special object which represents a promise of a value to be returned by a remote computation. It has the following fields:

value: When the future has a value, it is placed in this field.
msg-id: The unique id of the message which associated with the computation which will return a value to this future.
waiting-processes: The number of processes waiting for the future to have a value.
waiting-process-list: The list of processes waiting for the future, in **tconc** format.
single-assignment: A boolean field; true if the future can only be assigned a value once.
original-message: The contents of the **request-message** message sent to start the remote computation which will return a value to this future. Used when a clocked, **single-assignment** future is reposted.
destinations: The destination agents to which the original message was sent; used by **repost**.

multi-future

[Structure]

A **multi-future** is a collection of futures. It is returned by the **value-desired**, **multipost-style** messages. A **multi-future** contains a lists of satisfied and unsatisfied futures. Initially, all futures in a **multi-future** are unsatisfied; as values of remote computations are received, unsatisfied futures are given values and moved to the list of satisfied futures.

B.6 Declaring CAOS Agents

The file `czardecl.lisp` contains routines to declare sites and agents.

defsite

[Macro]

This macro makes it possible to declare Flavor classes which implement site-global storage within CAOS. `defsite` is defined in terms of `defherb`, and thus, it is possible to define instance variables within site instances which support abstract data type operations.

It is conceivable that if CAOS were ever implemented on a heterogeneous array of processors, there would be a number of site types, perhaps defined in a taxonomy.

vanilla-site

[Site]

Instances of `vanilla-site` implement site global storage. Each instance has the following instance variables:

static-agent-stream-table: Contains a dictionary which maps static (named) agents to their input stream addresses.

unresolved-agent-stream-table: Contains a dictionary which maps the names of remote agents not yet known during initialization to the addresses of streams in local agents which have requested the addresses of the unknown remote agent.

local-agents: A dictionary which maps the names of local agents to their addresses.

free-process-queue: A queue which holds information allowing free processes to be reused in preference to creating new processes.

care-site: Holds a pointer to the CARE site structure for the site upon which the `care-site` is located.

locale: Holds a CARE-defined structure which is created by `make-locale`, and which is updated by `update-locale`. Each call to `update-locale` modifies the structure so that a call to `locale-site` returns the least-recently-referenced site in the locale. This is a simple approach to load-balancing.

incoming-stream: Holds the stream upon which the site manager listens for site-oriented requests.

defagent-keyword

[Macro]

This macro defines the syntax for a new keyword used in a call to `defagent` (see below). The keywords described in Chapter 4, plus a number of keywords not described, are all declared through the use of `defagent-keyword`.

defagent

[Macro]

The `defagent` macro, which is defined in terms of `defherb`, is the basic form by which new agents are declared. It is described in detail in Chapter 4.

defagent-method

[Macro]

The **defagent-method** macro is syntactic sugar for **defmethod**, but has the advantage of being able to define the same method for multiple message names.

clock

[Abstract Data Type]

The **clock** ADT responds to the **:rearm**, **:tick**, and **:stop** messages. The **value** field of a **vp-slot** of the **clock** datatype holds a list of messages to be executed when the clock "fires."

vanilla-agent

[Agent]

The **vanilla-agent** is the most basic agent in the system. It has the following instance variables:

local-process-stream-table: A dictionary which maps from a process handle to a utility stream the process uses to wait for wakeup messages.

outstanding-message-table: A dictionary which maps from ids of messages to their associated futures.

runnable-process-list: A priority queue which implements the scheduling policy defined for the agent.

scheduler-lock: A monitor data type which is used to implement mutual exclusion around routines which modify the agent scheduler database.

process-table: A dictionary which maps from CARE process handles to CAOS runnable-items.

self-address: The stream upon which the agent's input process listens for requests and responses from other agents.

priority-queue-context: Holds information for creating nodes in the **runnable-process-list** priority-queue.

care-site: Points to the **care-site** structure for the site upon which the agent is located.

symbolic-name: Holds the name of the agent. Statically-created agents are named by the application program; dynamically-created agents are named by CAOS, using **gensym**.

agent-scheduler: Holds the CARE process handle of the process which is currently performing the duties of the agent scheduler.

running-processes: Holds a list of **runnable-item**'s which represent processes handed off to CARE for execution.

symbolically-referenced-agents: Holds a list of other agents to be referenced by name by methods executing within the context of the agent.

initial-forms: A list of expressions to be evaluated after CAOS has been initialized.

The purpose of these forms is to initialize an application.

:select-process-fifo *item-1 item-2* *[Method of vanilla-agent]*

This method implements FIFO scheduling of tasks within an agent. It is called as the priority function for the **priority-queue** stored in the **runnable-process-list**. Priorities are derived by comparing the **time-stamp** fields of *item-1* and *item-2*, which are **runnable-item's**.

process-agenda-agent *[Agent]*

The **process-agenda-agent** is a subclass of **vanilla-agent**. It differs from **vanilla-agent** in that certain message names may be given execution priorities. Such priorities are defined by specifying message names in order in a list stored in the **process-agenda** instance variable, messages at the front of the list have higher priority than those at the end of the list.

:select-process-agenda-timestamp *item-1 item-2* *[Method of process-agenda-agent]*

This method implements "agenda-based" scheduling of tasks in an agent. It is the priority function for the **runnable-process-list**. Priorities are derived by first comparing the **message-name** fields of *item-1* and *item-2*; if these fields are the same, the function then compares the **time-stamp** fields, as in the FIFO scheduler above.

B.7 Initializing a CAOS Application

The file **czarinit.lisp** contains the code which initializes CAOS at the start of a run. Initialization occurs in two distinct phases: one, *static*, before the CARE simulator is started, and the other, *dynamic*, just after.

The first set of functions, macros, and methods in **czarinit.lisp** is involved in static initialization. During this phase, the application initialization file (see Figure 4.4 and Appendix A) is read and interpreted. As a result of interpreting this file, all statically-declared agents are created on the appropriate sites, and the messages which initialize the application once CAOS is running are stored away.

:init *[after Method of care-site]*

During the static phase, new instances of **care-site** Flavor instances are created. The **:init** method is primarily responsible for initializing all of the abstract data types which are part of the **care-site**.

:init *[after Method of vanilla-agent]*

When a new agent instance is created, the **:init** method initializes a number of abstract data types, and also adds an entry to the appropriate **care-site's** **local-agents** dictionary.

make-initial-agent *agent-class global-name care-site*

[Macro]

This macro is invoked when the **caos-initialize** form is interpreted. *Agent-class* is the name of an agent class as defined by **defagent**. *Global-name* is the name by which this instance of the agent class will be known throughout the processing grid. *Care-site* is a two-element list specifying the *x* and *y* coordinates of the **care-site** upon which the new agent will be created. When the macro is executed, an instance of *agent-class* with name *global-name* is created on *care-site*.

initial-agent-record

[Structure]

This structure defines the a three-tuple with fields **name**, **class**, and **location**. Instances of this tuple make up the *agent-instances* argument to the **caos-initialize** macro (below). The **initial-agent-record** also defines the argument list to **make-initial-agent**.

caos-initialize *agent-instances initial-messages*

[Macro]

Calls to this macro are the means by which CAOS applications are initialized. *Agent-instances* is a list of **initial-agent-record** structures. *Initial-messages* is a list of expressions to be evaluated when CAOS has finished initializing.

When a **caos-initialize** form is evaluated, four major activities occur.

1. All statically-declared agents are created by mapping over *agent-instances* and calling **make-initial-agent** on each element.
2. An agent of class **initial-agent** is defined. The **initial-agent** class is a subclass of **vanilla-agent** which makes reference to all other statically-declared agents.
3. An instance of the **initial-agent** class, called 007 is created on site (1, 1).
4. The *initial-messages* argument is used to define an **:initial-form** method for the class **initial-agent**.

The remainder of **czarinit.lisp** is devoted to dynamic initialization. The necessary site and agent instances were created during the static phase; during the dynamic phase, these structures must be linked up with CARE. Dynamic initialization consists of starting the site manager processes in each of the sites, starting the input monitor and scheduler processes in each of the agents, and exchanging the names and addresses of each of the agents in order to resolve symbolic references. Dynamic initialization is completing by sending agent 007 an **:initial-form** message.

start-czar *initializer-stream*

[Process]

The **start-czar** process is the first process run once CARE starts. It drives all dynamic initialization tasks, as follows:

1. Creates a site manager process in each site.
2. Waits for each site manager process to return the address upon which it listens for requests.

3. Creates a process on each site that contains a statically-declared agent, whose task is to initialize those agents.
4. Waits for each site containing statically-declared agents to indicate its agents are initialized.
5. Sends the `:initial-form` message to the agent named 007.

start-site *initializer-stream site-stream*

[Process]

This process is the CAOS site manager. Upon start-up, it sends the value of *site-stream* to *initializer-stream* (upon which the *start-czar* process is waiting). It then enters an endless loop in which it responds to service requests directed to *site-stream*. The specific services implemented by the site manager were discussed in Section 5.2.

start-agents *all-care-sites-list start-agents-stream*

[Process]

This process is responsible for initializing statically-declared agents on each site. For each agent, it does the following:

1. Starts the input monitor process.
2. Broadcasts a `:new-initial-agent-online` message, containing the agent's name and the address upon which its input monitor process listens, to all other site managers in the grid (the value of *all-care-sites-list*).
3. For each agent named in the agent's *symbolically-referenced-agents* instance variable, sends a `:request-symbolic-reference` message to the site manager, and waits for a response.
4. Sends a message to the *start-czar* process indicating that the site is ready to run.

B.8 The CAOS Runtime System

The file *czar.lisp* contains the "runtime" system for CAOS. The functions documented in sections 4.3 and 4.4 are implemented by in this file. In what follows, we document those functions upon which the functions in these sections depend.

agendize *future*

[Defun-Method of vanilla-agent]

This is the low-level function used to suspend a process until *future* receives a value. It sets the calling process's state to `:suspended`, adds the process's *runnable-item* to the list of processes waiting for *future*, sets the context field of the *runnable-item* to be the process's wakeup stream, and sends to itself the `:reschedule` message, which invokes the scheduler to put the process to sleep. Upon waking up, it sets the process's state to `:running`, and returns to its caller (typically, *value-future*).

multi-agendize *multi-future*

[Defun-Method of vanilla-agent]

This function is the multi-future version of *agendize*.

remote-address-enumerating-functions

[Variable]

This variable holds an association list which maps ZETALISP data types into a function, which when applied to an object of the associated type, returns a list of remote addresses. This allows application programs built on top of CAOS to represent collections of agents in forms other than lists.

coerce-destination *dest-stream*

[Defun-Method of vanilla-agent]

This function coerces *dest-stream*, which may be a remote address, a future, or the name of an instance-variable in *self* into a stream.

If *dest-stream* is a remote-address, it is returned unmodified. If *dest-stream* is a symbol, it is evaluated in the context of *self*, and is expected to evaluate to a remote-address (this is the mechanism by which application programs are able to refer to statically-declared agents by name). Finally, if *dest-stream* is a future, *coerce-destination* calls *value-future* to retrieve the destination remote-address.

list-of-remote-addresses *list*

[Defun-Method of vanilla-agent]

This is the enumerating function for lists of remote addresses.

enumerate-destinations *remote-addresses*

[Defun-Method of vanilla-agent]

This function uses **remote-address-enumerating-functions** to coerce *remote-addresses* into a list of remote-address's.

stream-send *dest-stream priority flags message args*

[Defun-Method of vanilla-agent]

This function is a common subfunction used by CAOS-defined posting operators. It uses the facilities of CARE to send *message* and *args* to *dest-stream* with CARE priority *priority*. *Flags* is a list which controls the operation of *stream-send*. The following symbols may be included in *flags*:

:no-return —Causes *stream-send* to send a side-effect message.

:return-future —Causes *stream-send* to create a future, assign it a unique identifier, send the message with *self-address* as the return address, and return the new future to the caller.

:return-multi-future —Like *:return-future*, but causes *stream-send* to create and return a multi-future instead of a future.

:single-assignment —Causes *stream-send* to create a *single-assignment* future, a future whose value can only be set once.

make-and-initialize-future *type*

[Defun-Method of vanilla-agent]

This function creates a new future of type *type* (either *future* or *multi-future*). It also generates a unique identifier for the future in the agent's *outstanding-message-table*, and places the future in the table, keyed by the unique identifier.

format-stream-request *id stream message args*

[Function]

This function formats a message and its arguments for transmission to another agent. *Id* is the unique id of the message; *stream* is the stream to which answers should be directed.

agent-input-process *agent request-stream*

[Process]

This process is the process which monitors *self-address* for requests and responses from other CAOS agents. It is created exactly once per agent, and performs the following initialization steps:

1. Sets *self-address* to the value of *request-stream*.
2. Creates the agent scheduler process.
3. Arms all clocks in the agent.

After initializing the agent, *agent-input-process* enters a loop, in which it waits for messages directed to *self-address*, and then processes them accordingly.

:handle-request *request for-effect*

[Method of vanilla-agent]

This method is invoked by the input monitor process when a request message is received. It allocates a new *runnable-item*, and fills in its fields by copying from *request*, a *request-message* structure.

It then sends the new *runnable-item* to the scheduler process. If the scheduler is idle when this method is invoked, the *runnable-item* is sent to the process in a CARE message (this reawakens the idle scheduler); otherwise, the *runnable-item* is simply enqueued on the agent's *runnable-process-list*.

:handle-response *response*

[Method of vanilla-agent]

This method is invoked when the input monitor process encounters a *reponse-message*. It first checks if the response is directed towards a *future* or a *multi-future*. In the latter case, it calls upon the *:handle-multi-reponse* method to process the response. In the former case, it does the following:

1. If the future associated with the response is a single-assignment future, the future is removed from the agent's *outstanding-message-table*.
2. The value is removed from the response, and placed in the *value* field of the future.
3. The *satisfied* field of the future is set to *t*.
4. The *:run-processes* method is invoked, which restarts all processes waiting on the future.

`:handle-multi-reponse multi-future value source`

[Method of vanilla-agent]

This method is called when a response to a multi-future is received. *Source* is a cons of the sending agent's name and self-address; individual future's in the multi-future may be keyed by either.

The method uses *source* to find the appropriate future in the multi-future's unsatisfied-future list, and places *value* in its value field. If the multi-future is in `:any` wakeup mode, all processes waiting on the future are reawakened; if the multi-future is in `:all` mode, the waiting processes are reawakened only if there are no more unsatisfied future's.

`agent-scheduler agent scheduler-process-stream`

[Process]

This process is the CAOS scheduler process for agents. It is written as a loop which performs the following operations:

1. If the scheduler has previously determined that there are no runnable processes, or if there are requests waiting in the `runnable-process-stream`, the scheduler tries to get the next request from the `runnable-process-stream`. If neither condition is true, the scheduler skips to step 3, below.
2. If the message is a symbol, it is the name of a clock which has just ticked; in this case, the scheduler sends the `:tick` message to the clock.
If the message is a `runnable-item`, it is a request to the scheduler to perform an operation on the associated process. To be sent to the scheduler, the state of the process must be either `:suspended` or `:never-run`. In either case, the scheduler adds the item to the `runnable-process-list`.
3. The scheduler next tries to hand to CARE for execution as many processes as it can. The number of processes it is allowed to run at any one time is determined by the value of `*number-of-running-agent-processes*`.
4. Finally, the scheduler checks to see if any special conditions are outstanding. One special condition is that the user has requested a breakpoint (e.g., to perform some debugging with the CARE clock shut off). The other special condition is that it is about to be too late to perform an immediate garbage collection; in this case, the scheduler shuts off the CARE clock, and calls `gc-immediately`, the ZETALISP function which invokes the garbage collector.

`:add-to-runnable-process-list item`

[Method of vanilla-agent]

This method enqueues a `runnable-item` on the agent's `runnable-process-list`. If the CAOS instrumentation package is enabled, it also adds a line representing the process to the `scrolling-text-panel`.

`:choose-next-runnable-item`

[Method of vanilla-agent]

This method removes the highest-priority *runnable-item* from the *runnable-process-list*, unless the number of processes already handed to CARE is greater than or equal to **number-of-agent-running-processes**.

If the CAOS instrumentation package is enabled, and an item was removed from the queue, this method also removes the line representing the process from the *scrolling-text-panel*.

:schedule-next-process *return-new-items* [Method of vanilla-agent]

This method is called by the scheduler process to hand the highest-priority process to CARE for execution. If the state of the process is *:never-run*, the *:create-new-process* method is invoked to create a new process. If the state of the process is *:runnable*, the process is reawakened by calling the function *resume-old-item*.

:reschedule *future* [Method of vanilla-agent]

This method is invoked to suspend a process until *future* has a value. It first updates the CAOS instrumentation, then tries to run as many processes as possible (to keep the processor as busy as possible), and then suspends, waiting for a packet on its wakeup stream. Upon reawakening, it updates the CAOS instrumentation once again, and returns to its caller (typically *agendize*).

:create-new-process *runnable-item* [Method of vanilla-agent]

This method is called to create a new application-level process. It preferentially recycles a process waiting in the *free-process-queue* of the *care-site* associated with the agent. If there are no free processes available, it creates a new process using the facilities of CARE.

message-handler *agent runnable-item wakeup-stream* [Process]

All CAOS postings are executing in processes in which *message-handler* is the top-level. This process is a loop, which does the following:

1. Executes the message and arguments contained in *runnable-item*, an instance of a *runnable-item*.
2. Tries to pull the next *runnable-item* in state *:never-run* off the *runnable-process-list*. If there is such an item, *message-handler* returns to step 1 with *runnable-item* set to the new *runnable-item*.
3. Otherwise, the process queues itself on the *free-process-queue* of its associated *care-site*, to be reused later. It does this by calling the function *wait-for-an-item*.

czar-initialize *dimensions file aux-display* [Function]

This function is called to start CAOS. It initialize a number of global variables, sets up the CAOS instrumentation, and reads the *file*, the application file which contains the *caos-initialize* form.

Bibliography

- [1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structure and Algorithms*. Addison-Wesley, 1983.
- [2] N. C. Aiello, C. Bock, H. P. Nii, and W. C. White. *Joy of AGE-ing*. Technical Report, Heuristic Programming Project, Stanford University, 1981.
- [3] H. Brown, C. Tong, and G. Foyster. PALLADIO: An Exploratory Environment for Circuit Design. *IEEE Computer*, 16, December 1983.
- [4] H. I. Cannon. *Flavors: A Non-Hierarchical Approach to Object-Oriented Programming*. Technical Report, A.I. Lab, Massachusetts Institute of Technology, 1981.
- [5] B. A. Delagi. *The CARE User Manual*. Technical Report, Knowledge Systems Laboratory, Stanford University, 1986. In preparation.
- [6] Denelcor, Inc. *Heterogeneous Element Processor: Principles of Operation*. February 1981.
- [7] L. D. Erman, F. Hayes-Roth, V. R. Lesser, and D. R. Reddy. The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty. *ACM Computing Surveys*, 12:213-253, June 1980.
- [8] R. P. Gabriel and J. McCarthy. Queue-Based Multiprocessing Lisp. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, August 1984.
- [9] R. H. Halstead, Jr. Implementation of MultiLisp: Lisp on a Multiprocessor. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, August 1984.
- [10] B. W. Lampson and D. D. Redell. Experience with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105-117, February 1980.
- [11] V. R. Lesser and D. D. Corkill. The Distributed Vehicle Monitoring Testbed: A Tool for Investigating Distributed Problem Solving Networks. *The AI Magazine*, 15-33, Fall 1983.
- [12] E. Y. Shapiro. *Lecture Notes on the Bagel: A Systolic Concurrent Prolog Machine*. Technical Memorandum TM-0031, Institute for New Generation Computer Technology, November 1983.

- [13] R. G. Smith. *Structured Object Programming in Strobe*. Technical Report SYS-84-08, Schlumberger-Doll Research, March 1984.
- [14] R. G. Smith and P. Friedland. *Unit Package User's Guide*. Technical Report HPP-80-28, Heuristic Programming Project, Stanford University, December 1980.

Design and Implementation of a
Distributed Directory
Cache Coherence Protocol

Manu Thapar and Bruce Delagi

Department of Electrical Engineering
Stanford University
Stanford, CA 94305

and

High Parallel Systems and Applications
Digital Equipment Corporation
Palo Alto, CA 94301

Design and Implementation of a Distributed Directory Cache Coherence Protocol

Manu Thapar and Bruce Delagi
Stanford University
Digital Equipment Corporation

Abstract

This paper describes the design of a new hardware solution for the cache coherence problem in large scale shared memory multiprocessors. The protocol is based on a linked list of caches and does not require a global broadcast mechanism making our solution scalable. Directory based solutions proposed earlier also do not require a global broadcast mechanism. However, our *distributed directory protocol* has a lower cost and potentially better performance than the fully mapped directory based protocol. We do not assume that the network preserves the order of messages and allow adaptive routing. Our solution also allows an efficient implementation of locks. This paper describes the design and implementation details of the distributed directory protocol. Performance issues will be covered in a future paper.

1 Introduction

Cache coherence is an important well known problem in shared memory multiprocessor systems. If multiple caches are allowed to simultaneously have copies of a given memory location, a mechanism must exist to ensure that all copies remain consistent when the contents of that memory location are modified. The cache coherence problem may be solved through hardware or software. Hardware based protocols to solve the cache coherence problem are well understood for bus based shared memory architectures. Bus based hardware cache coherence protocols are also called snoopy cache coherence protocols [3]. The term "snoopy" is derived from the fact that each cache must watch all traffic on the bus and take appropriate action for addresses that are present in the cache. However the shared bus limits the number of processors that can be connected to the bus without saturating it. To

2 DIRECTORY BASED SOLUTIONS FOR THE CACHE COHERENCE PROBLEM

support scalable shared memory architectures, the cache coherence protocol needs to be able to work in the absence of a global broadcast mechanism. Directory based schemes [1, 7] are a possible solution in this environment. We present some drawbacks of these schemes in section 2.

This paper describes a new distributed directory cache coherence protocol. The information about which caches have copies of the data is decentralized and distributed among the cache lines. Our implementation, like the fully mapped directory scheme, tracks any number of cache copies and never requires invalidates to be sent to all caches in the system. It has a lower cost and potentially better performance than the fully mapped directory based coherence scheme for expected memory and cache sizes. In the fully mapped scheme, the size of the memory required to hold the state information is $O(MN)$, where M is the size of main memory and N is the number of caches. In our scheme, on the other hand, the size of the memory required to hold the state information is only $O(M \log N)$. We allow adaptive routing (so that network performance may be more robust) and do not assume that the network connecting caches preserves the order of messages. The network traffic generated for invalidations is reduced by a factor of two in our implementation compared to a fully mapped directory scheme for networks that do not preserve the order of messages. An important feature of this protocol is that locks can be supported very efficiently with minimal extra cost.

This paper is organized as follows. Section 2 describes the current directory based solutions for cache coherence problem. Section 3 describes our distributed directory cache coherence protocol. Section 4 talks about replacement of lines and potential race conditions. In section 5 we consider network issues, potential deadlocks and their handling. Section 6 describes how we intend to validate the correctness of our protocol. Section 7 describes the implementation of locks and section 8 states our conclusions and future work.

2 Directory Based Solutions for the Cache Coherence Problem

We will assume a very general architecture in our discussions. Figure 1 describes the basic architecture. Each node consists of one or more processing elements (P), a cache (C), a network controller (NC) and part of the distributed global memory (DGM). For the distributed directory protocol we do not assume that the network preserves the order of messages. This allows adaptive routing making the network performance more robust. The directory based protocols described in the literature so far do not allow out of order message arrival.

In the directory based protocols there is a directory "tag" associated with each line in main memory. This directory is used to hold information about which caches have copies of the line. In the fully mapped centralized directory scheme [7], the directory has N valid (or

2 DIRECTORY BASED SOLUTIONS FOR THE CACHE COHERENCE PROBLEM

support scalable shared memory architectures, the cache coherence protocol needs to be able to work in the absence of a global broadcast mechanism. Directory based schemes [1, 7] are a possible solution in this environment. We present some drawbacks of these schemes in section 2.

This paper describes a new distributed directory cache coherence protocol. The information about which caches have copies of the data is decentralized and distributed among the cache lines. Our implementation, like the fully mapped directory scheme, tracks any number of cache copies and never requires invalidates to be sent to all caches in the system. It has a lower cost and potentially better performance than the fully mapped directory based coherence scheme for expected memory and cache sizes. In the fully mapped scheme, the size of the memory required to hold the state information is $O(MN)$, where M is the size of main memory and N is the number of caches. In our scheme, on the other hand, the size of the memory required to hold the state information is only $O(M \log N)$. We allow adaptive routing (so that network performance may be more robust) and do not assume that the network connecting caches preserves the order of messages. The network traffic generated for invalidations is reduced by a factor of two in our implementation compared to a fully mapped directory scheme for networks that do not preserve the order of messages. An important feature of this protocol is that locks can be supported very efficiently with minimal extra cost.

This paper is organized as follows. Section 2 describes the current directory based solutions for cache coherence problem. Section 3 describes our distributed directory cache coherence protocol. Section 4 talks about replacement of lines and potential race conditions. In section 5 we consider network issues, potential deadlocks and their handling. Section 6 describes how we intend to validate the correctness of our protocol. Section 7 describes the implementation of locks and section 8 states our conclusions and future work.

2 Directory Based Solutions for the Cache Coherence Problem

We will assume a very general architecture in our discussions. Figure 1 describes the basic architecture. Each node consists of one or more processing elements (P), a cache (C), a network controller (NC) and part of the distributed global memory (DGM). For the distributed directory protocol we do not assume that the network preserves the order of messages. This allows adaptive routing making the network performance more robust. The directory based protocols described in the literature so far do not allow out of order message arrival.

In the directory based protocols there is a directory "tag" associated with each line in main memory. This directory is used to hold information about which caches have copies of the line. In the fully mapped centralized directory scheme [7], the directory has N valid (or

3 THE DISTRIBUTED DIRECTORY CACHE COHERENCE PROTOCOL

the network traffic is lower¹. However, if the buffer overflows, the requests still have to be bounced back. Requiring transactions to be serialized through the centralized directory (and the locking of lines while servicing a request that requires a coherency-related transaction) could make the directory a bottleneck.

To reduce the amount of storage required, a number of modifications to the above scheme may be made. However, these modifications either require the implementation of an efficient broadcast mechanism contradicting our assumption about scalable systems, or may generate excess network traffic along with performance penalties. For example, one simple modification is to have i pointers per line in the directory. Each pointer may point to a cache that has a copy of the line. If more than i caches have copies of the line, a broadcast has to be done to *all* caches to service a write miss. The memory line has to be locked until all caches acknowledge the invalidation. This is classified as a Dir_iB scheme [1], where i is the number of indices kept in the directory and B stands for broadcast. A Dir_iNB scheme, where i is less than the number of caches and NB stands for no broadcast, is possible also. In such a scheme, at most i caches can have copies of a line at the same time. In the case where a read miss occurs when i caches have copies of the line, the directory has to invalidate one of the copies before the data can be supplied to the requesting cache. This might result in "thrashing" the line between caches.

3 The Distributed Directory Cache Coherence Protocol

Based on a linked list of caches [6], we now describe the distributed directory protocol. We shall first provide the basic idea of the protocol and then explain it in further detail. Each line in the main memory and the cache has a cache-pointer field associated with it. This pointer can address any cache in the system. The directory services read or write request by changing cache-pointer in the directory entry associated with the line to point to the requesting cache. If the old value of cache-pointer is nil, a proper reply is sent to the requesting cache. If the old value of cache-pointer points to a cache, the request is forwarded to that cache. In case of read misses the cache replies to the requesting cache and the list now includes the requesting cache. In case of write misses, the list has to be invalidated before a reply can be sent to the requesting cache.

The amount of memory required for the pointer is $\log N$ where N is the number of caches. The total amount of memory needed is thus $O(M \log N + Nc \log N)$ where M is the total size of main memory, N is the number of caches and c is the size of each cache. The above

¹If there is contention at the directory and the network is lightly loaded, then a buffer may *degrade* the performance instead of improving it. This is because the directory controller would have to spend time managing the buffer and thus further increase the contention at the directory.

3 THE DISTRIBUTED DIRECTORY CACHE COHERENCE PROTOCOL

expression can be written as $O(M(1+k)\log N)$ where k is Nc/Nm (m being the amount of memory per node). We interpret k as the ratio of the size of cache memory per node to the size of main memory per node.

Assuming a constant value of k for the machine, the amount of memory required for the distributed directory scheme is $O(M\log N)$. We can expect then, that the cost of implementing the distributed directory scheme is significantly less than the fully mapped scheme—which requires $O(MN)$ amount of memory.

3.1 The Protocol

Cache coherence protocols can be effectively explained using state diagrams. Each line in the cache has a *local state* associated with it. Similarly, each line in main memory has a *global state* associated with it. The local states and the global states are used by the cache controller and memory controller respectively to take appropriate action. Memory requests issued by the processors may be serviced by their caches. In case of cache misses, the cache controller issues requests to the appropriate main memory module. A reply is subsequently received, either from the main memory, or from another cache. The communication between the caches and the main memory may be considered to be similar to asynchronous message passing. A read miss occurs when the cache does not have a valid copy of the line. A write miss occurs when the cache does not have the line in state 'exclusive'. The cache controller sends a read-miss 'RM(c)' signal or a write-miss 'WM(c)' signal to the main memory on read and write misses. Before a miss signal is issued by a cache controller, a line associated with the address is allocated in cache memory. The miss signals include the address of the source of the request '(c)'.

Figure 2 shows the global state diagram for the distributed directory protocol. Table 1 shows the actions taken by the main memory to respond to the signals. A line in main memory is originally in state absent (A) (since it is absent from all caches). Each request causes the value of cache-pointer to be updated to point to the requesting cache. If the line is absent from all the caches, the main memory sends a reply and changes its state to present (P) (since it is now present in at least one cache). Otherwise the request is forwarded to the last cache to make a request for the same line. If the line is absent from all caches, the directory sends a write-miss-reply 'WMR(d)' or a read-miss-reply 'RMRD(d)' for a write-miss or a read-miss signal respectively. A read-miss-reply 'RMRD(D,d)' consists of the data (d) and the address (D for Directory) of the object replying to the request. A write miss reply 'WMR(d)' consists of the data. In directory based protocols, special care has to be taken to handle races properly. The state R is used to handle races properly. We defer the discussion about races to section 4.

Figure 3 shows the local state diagram for the distributed directory protocol. Table 2 and table 3 show the actions taken by the main memory to respond to the signals. A line

3 THE DISTRIBUTED DIRECTORY CACHE COHERENCE PROTOCOL

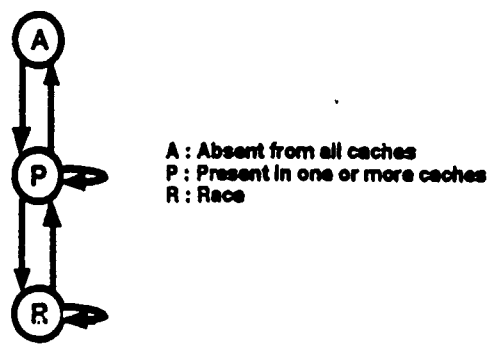
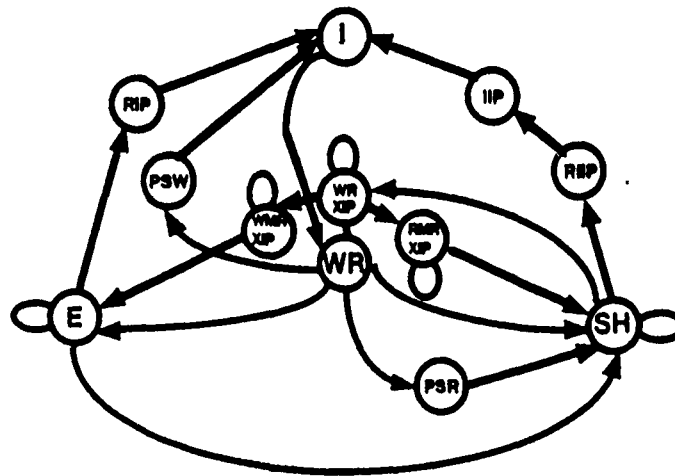


Figure 2: Global State Diagram

State	Signal	Next State	Actions
A	RM(c)	P	p=c; RMRD(d) to c
A	WM(c)	P	p=c; WMR(d) to c
P	RM(c)	P	Old-p = p p=c; RMR(D,d) to c
P	WM(c)	P	Old-p = p p=c; WMR(d) to c
P	Re(c,d); c=p	A	mem = data
P	Re(c,d); c!=p	R	mem = data
P	RMFB(c)	R	mem = RMR(c)
P	WMFB(c)	R	mem = WMR(c)
R	RMFB(c)	P	RMR(D,d) to c
R	WMFB(c)	P	WMR(d) to c
R	Re(c,d); c!=p	P	signal = mem mem = data service signal

Table 1: Global State Transitions.

3 THE DISTRIBUTED DIRECTORY CACHE COHERENCE PROTOCOL



I : Invalid
 E : Exclusive
 SH : Shared
 WR : Writing or Reading
 PSW : Pending Signal Write
 PSR : Pending Signal Read
 X : Replace or Invalidate
 IP : In Progress

Figure 3: Local State Diagram For Main States

is originally in state invalid (I). A read 'Rd' or a write 'Wr' request from the processor causes the state to change to writing-or-reading 'WR' and a proper signal to be sent to the appropriate main memory module as show in table 2. On a read-miss-reply, the value of cache-pointer is set to be the address of the object sending the reply. This causes a linked list of caches that contain the data in shared state to be formed.

Figure 4 illustrates the process followed to set up the linked list. Consider the case where cache C1 has a read miss for a line followed by caches C2 and C3. As show in fig. 4(a), cache C1 sends a read-miss signal to the directory. The cache-pointer of the line in the directory is made to point to C1. Since no other cache has a copy of the line, the main memory sends a write-miss-reply to C1. When C1 receives the reply the line is loaded into the cache in state "exclusive". Now when cache C2 sends a read-miss to the directory, a read-miss-forward 'RMF' signal is sent to C1 as shown in fig. 4(b). The cache-pointer in the directory now points to C2. When C1 receives the forwarded signal it changes its state to "shared" as

3 THE DISTRIBUTED DIRECTORY CACHE COHERENCE PROTOCOL

State	Signal	Next State	Actions
I	Rd	WR	RM(c)
I	Wr	WR	WM(c)
I	IF(c)	I	IA to c
I	RMF(c,d)	I	RMFB(c) to dir
I	WMF(c,d)	I	WMFB(c) to dir
WR	RMRD(c,d)	E	Proc = data cache = data; p = c
WR	RMR(c,d)	SH	Proc = data cache = data; p = c
WR	WMR(d)	E	cache = proper data
WR	WMR-data(d)	WR-1	cache = proper data
WR	WMR-performed	WR-1	
WR-1	WMR-data(d)	E	cache = proper data
WR-1	WMR-performed	E	proc = data
WR	RMF(c)	PSR	PS = RMR(c)
WR	WMF(c)	PSW	PS = WMR(c)
E	Rd	E	Proc = reply
E	Wr	E	cache = data
E	RMF(c,d)	S	RMR(c,d)
E	WMF(c)	I	WMR(d) to c
E	Rc	RIP	Re(c,d)
SH or S	Rd	same	Proc = reply
SH	RMF(c)	S	RMR(c-self,d)
SH	WMF(c)	I	if (p=c) WMR(d) to c else WMR-data(d) to c WMFC(c) to p
S	WMFC(c)	I	If (p=nil or p = c) WMR-performed to c else WMFC(c,d) to p
SH	Wr	WRIIP	I(c) to p
SH	Rc	RIIP	I(c) to p
S; p != nil	Wr	WRXIP	WM(c) I(c) to p
S; p = nil	Wr	WR	WM(c)
S; p != nil	Rc	IIP	I(c) to p
S; p = nil	Rc	I	
S	I(c)	I	If p=nil IA to c else I(c) to p

Table 2: Main Local State Transitions. Part (a).

3 THE DISTRIBUTED DIRECTORY CACHE COHERENCE PROTOCOL

State	Signal	Next State	Actions
PSR	RMR(c,d)	S	Proc = data cache = data
PSR	WMR(d)	S	RMR(c-self, d) cache = proper data
PSW	RMR(c,d)	I	RMR(c-self,d) Proc = data
PSW	WMR(d)	I	WMR(d) cache = proper data
WRXIP	IA	WR	WMR(proper data) nil
WRXIP	WMR(d)	WIIP	cache = proper data
WRXIP	RMF(c)	PSRXIP	PS = RMR(c)
WRXIP	WMF(c)	PSWXIP	PS = WMR(c)
PSRXIP	WMR(d)	PSRIIP	cache = proper data
PSWXIP	WMR(d)	PSWIIP	cache = proper data
PSRIIP	IA	S	RMR(c-self, d)
PSWIIP	IA	I	WMR(d)
WIIP	IA	E	
WIIP	RMF(c)	PSRIIP	PS = RMR(c)
WIIP	WMF(c)	PSWWIIP	PS = WMR(c)
RIIP	IA	RIP	Re(c,d) to dir
RIP	RA	I	
IIP	IA	I	

Table 3: Main Local State Transitions. Part (b)

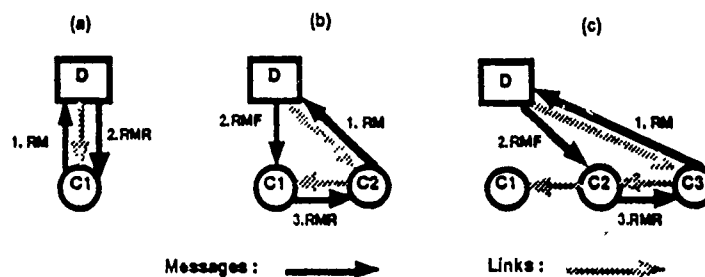


Figure 4: Linking of caches due to read misses

3 THE DISTRIBUTED DIRECTORY CACHE COHERENCE PROTOCOL

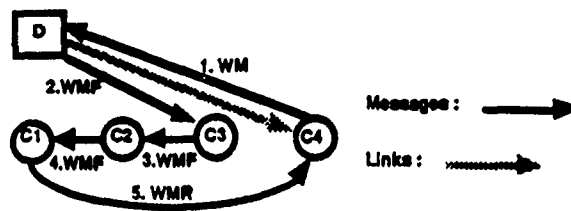


Figure 5: Invalidation due to write misses

shown in table 2. When C2 receives the reply, it sets its cache-pointer to point to C1 and loads the line in state "shared-head". Thus a linked list is formed. Fig. 4(c) shows how C3 gets linked into the list. When cache C3 has a read miss, the request is forwarded to cache C2 by the directory. The cache-pointer in the directory now points to C3. When C2 receives the forwarded signal it changes its state to "shared"², and sends the reply to C3. When C3 receives the reply, it sets its cache-pointer to point to C1 and loads the line in state "shared-head". In the local state diagram shown in figure 3 we have shown states "shared-head" (SH) and "shared" (S) as one state since they are quite similar and this simplifies the figure. C1 sends a reply to C2. The reply consists of the data and the address of C1.

Write misses cause a WM signal to be sent to the directory. Write buffering along with weak ordering [9] is used to allow the processor to proceed in the case of write misses. A write is considered to be *issued* when a write-miss is sent by the cache. A write is considered to be *performed* when a write-miss-reply is received by the cache. A *fence* [15] operation may be used to ensure that all writes that have been issued are performed before the processor is allowed to proceed. If a copy of the line is not present in any other cache, the directory can directly send a reply. Otherwise the copies of the line have to be invalidated before a reply can be sent. Figure 5 shows the sequence of events that result when multiple caches have a copy of the line. The directory forwards the write miss signal 'WMF' to the old head pointed to by cache-pointer and cache-pointer is updated to point to C4. When C3 receives the write-miss-forward signal, it invalidates its copy and forwards the signal to C2. C2 does the same and forwards the signal to C1. Since the cache-pointer of C1 points to nil, it can be determined locally that C1 is the tail of the list and a reply is sent to C4 after the data in C1 is invalidated.

Sometimes, due to replacement of lines it may not be possible to determine locally that a cache is at the tail of the list. The scenario under which this may happen will be explained in further detail in section 4 on replacements. We present the solution to this problem here.

²The distinction between "shared-head" and "shared" needs to be done locally since only the head of the list needs to send data to the main memory as discussed in section 4.

3 THE DISTRIBUTED DIRECTORY CACHE COHERENCE PROTOCOL

Consider the case that a cache is the current tail of the list but has a non-nil cache pointer (say C5) due to replacement of the line in C5. In this case a WMF signal is sent to C5. The cache controller at C5 notes that it does not have a copy of the line and sends a WMR to the requesting cache. Since C5 does not have a copy of the data, some means must be used to send the data to the requesting cache. One method may be to have the old head (C3) send the data along with the WMF signal. This would mean that a WMF has a cache line of data associated with it. This would result in performance degradation due to excess network traffic and other factors. A better method, that we use, is to let the old head (C3) send a write-miss-reply-data signal to the requesting cache along with the data. When the requesting cache receives this signal it stores the data and may service any read requests that are associated with this line and are allowed to be serviced. A write-miss-forward-cache signal traverses the distributed directory and the tail of the directory sends a write-miss-reply-performed signal to the requesting cache. The requesting cache has to receive both, the write-miss-reply-data and the invalidate-acknowledge signals to ensure that a write has been "performed". To keep the state diagram simple, we use a counter to ensure that both, the write-miss-reply-data and the write-miss-reply-performed signal are received. Thus there are three different kind of signals for write miss replies.

1. The write-miss-reply signal carries the data and implicit information that the write has been "performed". This is sent when the main memory directly sends the reply to the requesting cache. It is also sent by a replying cache if that cache is in state "exclusive" when it received the write-miss-forward signal.
2. The write-miss-reply-data signal carries the data. An explicit write-miss-reply-performed signal has to be received by the requesting cache before the write can be considered to be "performed". This is used by the head of the shared list to send the data to the requesting cache.
3. The write-miss-reply-performed signal is used to inform the requesting cache that all the caches that have copies of the shared data have been invalidated. This is normally sent by the tail of the shared list.

If the line in the cache is in state WR and a RMF or WMF signal is received, it is stored locally. The address of the requesting cache is stored in the cache-pointer field of the cache line and the state of the cache line is changed to note that a forwarded signal has been stored. As shown in table 3, the state is changed to pending signal read 'PSR', or pending signal write 'PSW' depending on the forwarded signal. These signals are serviced when the reply to the local read or write miss is received.

When the data is in shared state in the cache, a write (Wr) by the processor produces a write miss. In this case an invalidate is sent to the cache pointed to by the cache-pointer

4 RACES AND REPLACEMENT OF LINES

and a WM is sent to the appropriate directory. The local state is changed to "writing-reading-X-in-progress" WRXIP. The state changes to exclusive when both, an invalidate acknowledge (IA) and a WMR are received as shown in table 3. If the write-miss-reply is received before the invalidate-acknowledge signal, the intermediate state WIIP (writing-invalidate-in-progress) is used. If the reverse happens, the intermediate state WR is used. As we have seen, when the directory controller receives a write or read miss request, and the data is present in some cache(s), the request is forwarded to the head of the list. No locking is needed at the directory. The reply is sent by a cache directly to the requesting cache. This prevents the directory from becoming a potential bottleneck and overcomes the problems related with locking the directory mentioned in section 2.

4 Races and Replacement of lines

Line replacement can cause certain race conditions to arise [4, 16] in distributed directory protocols. In this section we will explain the mechanisms involved with line replacement and races for the distributed directory protocol. On replacement of a cache line, data needs to be transferred to the directory when the line in the cache is in state exclusive (E), or it is the head of the shared list (SH). For caches that are part of a shared list, the lower portion of the list is invalidated when a replacement is done. Caches that are part of a shared list, but not at the head of the list, do not have to send data to the directory on replacement.

If the state is exclusive, a replace signal 'Re(c,d)' is sent to the directory. The state changes to replace-in-progress (RIP). When the directory receives the replace signal, it copies the data and sends a replace-acknowledge to the cache. When the cache receives the replace-acknowledge signal, it changes the state of the line to invalid and can now use it for the address that caused the replace to occur.

A race may happen during this replacement. A write-miss-forward (or read-miss-forward) signal may be sent by the main memory at the same time as the replace signal is sent by the cache to the main memory. In order to allow adaptive routing, we do not require that the network preserve the order of messages. The forwarded signal may reach the cache when it is in state replace-in-progress or after the cache has received a replace-acknowledge signal in which case the cache no longer has a copy of the line. In either case, the cache bounces the signal back to the directory as a write-miss-forward-bounce WMFB (or a read-miss-forward-bounce) signal. The directory takes care of this race. When the main memory receives a replace signal, it checks to see if the value of the cache-pointer is the same as the address of the cache sending the replace. If it is, things are fine and the race did not occur. The data is copied to the main memory, and the state is changed to absent. If the cache-pointer has a different value, it means that a write-miss (or read-miss) was received by the directory and this was forwarded to the cache that sent the replace signal. When this happens, the data is copied to main memory and the state is changed to race. This is shown in figure 2

4 RACES AND REPLACEMENT OF LINES

and table 1. The forwarded signal comes back to the main-memory as a forward-bounced signal and is serviced by the main memory. The state is changed from race to present when the forwarded signal is received. Since the network need not preserve the order of messages, the bounced signal may be received before the replace signal is received. If the line in main memory is in state present when a bounced signal is received, it means that this race has occurred. The signal is stored locally and the state is changed to race. The signal can be stored in the memory line since the line does not have valid data. No extra storage is required to store this signal. This stored signal is serviced when the replace signal is received and the state is changed to present.

Now we consider the case when the cache is part of a shared list. There are two sub-cases. The cache may be at the head of the list in state SH, or it may be within the shared list in state S. We first consider the case when the cache is within the list. In this case, the data does not need to be sent to the directory. If the cache is at the tail of the list (its cache pointer is nil), the state can be changed to invalid and the line can be used immediately. Otherwise, the cache sends an invalidate-forward signal to the next cache in the list and changes its state to invalidate-in-progress IIP. The cache that receives an invalidate-forward signal invalidates its copy and forwards the signal to the next cache. The tail of the linked list sends an invalidate-acknowledge signal to the originating cache. When the invalidate-acknowledge signal is received by the originating cache, it changes its state to Invalid and the line can now be used for a different address. It should be noted here that it is not always possible to determine locally that a cache is the tail of the list. For example, when a cache line that is within the linked list is replaced, the cache that is towards the head does not know that it is the new tail. To take care of this, when an invalidate-forward signal is received by a cache that does not have a valid copy of the line, it sends an invalidate-acknowledgement signal to the originating cache.

More than one cache that are part of the shared list may want to replace their copies of the line at the same time. In this scenario, the caches that want to replace their lines send invalidate-forward messages to the next cache. An invalidate-forward message originating from a cache (C1) may be received by a cache (C2) while it is in state invalidate-in-progress. In this case, the invalidate-forward signal is stored locally at C2 and an invalidate-acknowledgement is sent to C1 when C2 receives an invalidate-acknowledgement for its own replacement.

Now we consider the case when the cache is the head of the shared list, i.e. its state is SH. In this case the data needs to be sent to the main memory on replacement. First, an invalidate-forward signal is sent to the next cache and the state is changed to replace-invalidate-in-progress. When the invalidate-acknowledgement is received, the state is changed to replace-in-progress and a replace signal is sent to the directory along with the data. The main memory follows the same procedure as described previously and sends a replace-acknowledge signal to the cache. The cache can then change the state of the line to invalid

5 NETWORK ISSUES AND DEADLOCKS

and use it for the address that caused the replace to occur.

Certain optimizations to this basic scheme are possible. For example, a fully associative *replacement buffer* may be used to buffer the lines that are in the process of being replaced. This would minimize the effect of replacement of lines. The size of this buffer is expected to be small.

5 Network Issues and Deadlocks

Any interconnection network may be used to connect the nodes to each other. Thus the network may any combination of a multistage interconnection network, a torus, a mesh etc. using store and forward routing, virtual cut-through routing, wormhole routing, adaptive routing etc. However, it should be ensured that the routing algorithm that is used is deadlock free. A number of deadlock free algorithms for various networks have been proposed in the literature [8, 12, 10].

Usually, to prove that a network is deadlock free, it is assumed that a message arriving at its destination node is eventually *consumed*. In directory based protocol this assumption is not necessarily true since when a message is received, another one may have to be sent before the message may be considered to be consumed.

In the distributed directory protocol we have to take care of the following kinds of deadlocks.

1. Request-reply deadlock: When a miss request is received by a directory, a reply has to be sent (assuming the line is absent from all other caches) to the requesting cache before the request can be considered to be consumed. This can produce a request-reply deadlock.
2. Request-request deadlock: When a miss request is received by a directory, a request may need to be forwarded (assuming the line is present in another cache) to another node before the request can be considered to be consumed. This can produce a request-request deadlock.
3. Reply-reply deadlock: When a reply is received by a cache, it is possible that a reply needs to be sent (assuming that there is a pending signal for the line) to another node before the reply can be considered to be consumed. This can produce a reply-reply deadlock.

To take care of these deadlocks, we use three *logical* networks along with time-outs. A request network is for the requests, a *reply-network* is used to send all the replies and an *exception-network* is used to send the exception retry signals on time-outs.

6 CORRECTNESS OF THE PROTOCOL

Request-reply deadlocks are taken care of by having different request and reply networks. The exception-network allows us to take care of request-request and reply-reply deadlocks.

Let us consider a request-request deadlock to explain the mechanism for time-outs. When a miss request is received by the directory, it needs to be forwarded to the head of the list if the line is present in any cache. Thus the request cannot be consumed before another request is sent out. This may generate cycles in the network resulting in deadlock. A time-out mechanism is used to handle such cases. On a time-out, an exception is sent to the originating cache (say C1) along with a modification of the original request. The directory follows the same protocol as it would if the request could have been successfully forwarded i.e. it changes its cache-pointer to point to C1 and changes the global state if required. The original request is modified to include the address of the cache (say C2) that the request would have been forwarded to. When C1 receives the exception signal, it now directly sends the request to C2.

6 Correctness of the Protocol

The correctness of cache coherence protocols is a very important issue. As cache coherence protocols become more complex, it becomes increasingly important to analyze the correctness of the protocol. We plan to validate the correctness of our protocol by building simulation models of the protocol and by using suitable test programs to drive the simulation models.

7 Efficient Implementation of Synchronization Variables

In shared memory architectures it is very important to handle locks and barriers efficiently. The distributed cache coherence scheme can be used to implement a very efficient scheme of locks at minimal extra cost. The implementation of locks allows us to provide an efficient software solution of barriers.

7.1 Distributed Locks

Most architectures have some form of atomic test&set instruction to implement spin locks [2]. The test&set instruction sets the value of a memory location and atomically returns the old value. When a process wants access to a lock, the processor performs the test&set instruction. If the operation is successful, the processor continues. Otherwise the processor repeatedly tries to access the lock until it is successful. This mechanism is known as spin lock or busy wait. Spinning on a test&set instruction can cause a lot of network traffic. The network

7 EFFICIENT IMPLEMENTATION OF SYNCHRONIZATION VARIABLES

```
procedure lock(var)
begin
    Queue-Lock(var)
    while (Test&Set(var))
        Queue-Lock(var)
end
```

Figure 6: Lock procedure

traffic due to busy wait is $O(N^2)$ where N equals the number of processors that are busy-waiting. This can be confirmed by noting that each time a lock is released, only one processor is successful in getting the lock, but each processor performs the test&set operation. Such implementations of locks can result in starvation and the accessing of locks is not fair.

The distributed cache coherence protocol allows an implementation of locks that makes the network traffic minimal. Lock requests are queued and normally serviced in FIFO order³. The network traffic is $O(N)$. Fine grain locking is provided by having a lock bit per cache-line [5]. The main advantage of providing a lock bit per cache line is that the data associated with the lock is obtained at the same time as the lock.

Figure 6 shows the code for a lock procedure. The queue-lock instruction is used to join a queue of nodes waiting for a lock. A cache line with the lock bit set is allocated on a queue-lock (QL) instruction. The test&set instruction returns a value of 'true' until the lock is obtained at the local cache.

The queue of locks is formed in a similar way as the linked list for caches that have shared copies of a line are formed. The implementation of locks requires a few extra states and locks. Figure 7 shows the global state diagram for lines using lock bits. Such lines are called *hard atoms* [5]. Table 4 shows the corresponding transition table. Figure 8 shows the local state diagram for hard atoms and table 5 shows the corresponding transition table.

When a queue-lock instruction is executed, a line in state lock-in-progress 'LIP' is allocated and a lock-miss 'LM' signal is sent to the directory. If no other cache has locked the line, the directory sends a lock granted 'LG' signal along with the data. The first cache that receives the lock-granted signal from the directory is considered as the upstream end. Otherwise the directory updates its cache-pointer to point to the requesting cache and forwards the lock-miss signal to the cache pointed to by cache-pointer previously. The cache that receives this forwarded signal, stores the address of the requesting cache (the *downstream* pointer) in its cache-pointer field. A set-upstream-pointer 'SUP' signal with the old value of

³The FIFO order may not be exactly preserved in the unlikely case that the cache line associated with the lock has to be replaced.

7 EFFICIENT IMPLEMENTATION OF SYNCHRONIZATION VARIABLES



Figure 7: Global states for hard atoms (locks)

State	Signal	Next State	Actions
A	LM(c)	L	p=c; LMR(D,d) to c
L	LM(c)	L	Old-p=p; p=c LMF(c) to Old-p SBP(Old-p) to c
L	R(c,d); c=p	A	mem=data
L	R(c,d); c!=p	L	LG-Next(c,d) to p
L	RL(c,d)	HR	mem=data UA to c
L	RM(c)	HR	mem=WMRL(c)
L	WM(c)	HR	mem=WMRL(c)
L	release-lock(c,d)	L	release-lock(d) to p
HR	LM(c)	HR	Old-p=p; p=c LMF(c) to Old-p SBP(Old-p) to c
HR	RM(c) or WM(c)	L	WMRL(d) to c
HR	RL(c,d)	L	signal=mem mem=data service signal

Table 4: Main global state transitions for hard atoms.

7 EFFICIENT IMPLEMENTATION OF SYNCHRONIZATION VARIABLES

State	Signal	Next State	Actions
I	QL(c)	LIP	LM(c)
I	T&S	I	return 1 to proc
LIP	LG(D,d) or release-lock(c,d)	LH	cache = data
LIP	SUP(c)	UP	up = c
LIP	LG(c,d) or release-lock(c,d)	LG	cache = data
LIP	LMF(c)	LIP	PS = LG(c); set DPP
LH	Test&Set	L	Proc = 0
LH	LMF(c)	LH	PS = LG(c); set DPP
LH	Rc	A2	Re(c,d) to dir
UP	LG(c,d) or release-lock(c,d)	LH cache = data	
UP	LMF(c)	UP	PS = LG(c); set DPP
UP	Rc	A1	UUP (c,bp) to dir UDP(c,nil) to BP
LG	SUP(c)	LH	nil
LG	LMF(c)	LG	PS = LG(c); set DPP
L	Unlock; DPP not set	LHU	nil
L	Unlock; DPP set	I	LG(c,d)
L	LMF	L	PS = LG(c); set DPP
L	Rc	A2	RL(c,d) to dir
L	T&S	L	return 1 to proc
LHU	Test&Set	L	Proc = 0
LHU	LMF(c)	I	LG(c,d)
LHU	Rc	A2	R(c,d) to dir
LF	Unlock	I	LG(c,d)
LF	Rc	A2	RL(c,d) to dir
A1	UA or	LMF(c)	A2 nil
A2	UA or LMF(c)	I	nil
WR	WMRL	LFX	
LFX	Unlock	I	release-lock to dir
X	release-lock(c,d);c != up	X	release-lock to up

Table 5: Local state transitions for hard atoms.

7 EFFICIENT IMPLEMENTATION OF SYNCHRONIZATION VARIABLES

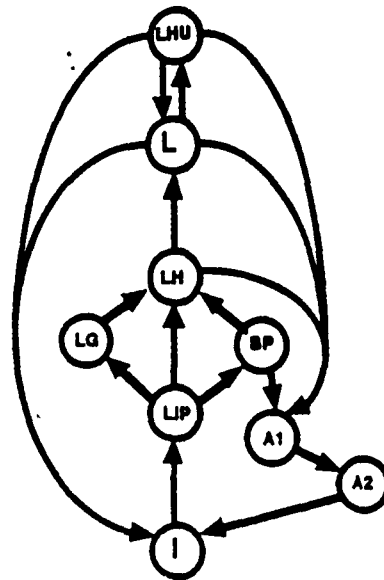


Figure 8: Local states for hard atoms

the cache-pointer is also sent to the requesting cache. The last requesting cache is considered as the downstream end. The directory points to the downstream end. The requesting cache stores this *upstream* pointer in its data field. In this way, a doubly linked list of caches waiting for a lock is formed. Lines that are in state "lock-in-progress" do not have valid data, so these lines can store the upstream pointers for the doubly linked list without requiring extra memory storage. The "lock-in-progress" state has a few *sister states* that are used to ensure that the signals required to form the doubly linked list are received. If the lock-granted signal is received first, the sister state LG is used. If the SUP signal is received first, the sister state UP is used. When both these signals are received, the state becomes lock-held 'LH' and the lock bit is unset. Now a Test&Set instruction can be completed successfully. More than one processor may share a cache and there may be more than one process contending for the same lock. When a lock-granted signal is received, a process has to obtain the lock and release it to allow other caches waiting for the lock to get the lock. If no other cache wants the lock, the line is held locally. Locking and unlocking of the line by processes that share the cache can now be done in the cache very efficiently without requiring any network traffic. After the lock has been used once locally and a lock-miss-forward signal is received, the lock is granted to the next cache in the queue. The grant signal does not have to go through the directory and the data is passed between the caches.

Unlocking of locked lines by processes other than the one that performed the lock is allowed. A unlock signal is sent to the directory which forward the signal to the tail of the queue of caches. The signal flows upstream until it reaches the head node with the locked

7 EFFICIENT IMPLEMENTATION OF SYNCHRONIZATION VARIABLES

line.

Modulo replacement, lock requests are serviced in FIFO order and starvation is eliminated if there is one process per cache. This solution only requires $O(N)$ operations. A somewhat similar scheme has been proposed by Goodman et.al. [11], however their implementation requires the interconnect to support global broadcasts and the transfer of the lock and data to the next cache in the queue requires interactions with the main memory, resulting in more network traffic.

Replacement of lines in caches that are queued for the lock is allowed by sending signals to the upstream and downstream caches to update their links and waiting for acknowledgements before the line is replaced. To avoid deadlocks, signals from the upstream side are given higher priority. The replacements is thus non destructive. The process associated with the line that got replaced issues a queue-lock instruction at a later time to join the queue again. If the line associated with the queue-lock instruction is not absent from the cache, the instruction does not cause any actions. Replacement of locked lines is done similarly by updating the backward link of the next cache to point to the directory. The data is copied to the main memory and the global state becomes HR implying that a held lock has been replaced. The process that held the lock is free to migrate. A subsequent read or write miss by the process causes the line to be loaded in locked state in the cache. When a locked line is replaced, the forward pointer is lost. In such cases, when the lock is released, a release-lock signal is sent to the directory which in turn forwards it to the cache pointed at by cache-pointer. The signal flows upstream until it reaches a cache whose back pointer points to the directory. The FIFO order of granting locks is then resumed. Processes that have issued the queue-lock instruction, but not yet obtained the lock successfully are allowed to migrate by making the operating system replace the lines in state lock-in-progress associated with the process that has to be migrated. This is done by maintaining a data structure that contains the address of lock lines requested by the process, but not locked by that process.

7.2 Barriers

Barriers are an important form of synchronization in shared memory multiprocessors. Execution of a thread of control cannot proceed beyond a barrier until a specified number of threads have reached the barrier. Most architectures would implement a barrier as follows. A shared variable `num-barrier` is initialized to the number of threads that must wait at a barrier. There is also a lock associated with the `num-barrier`. When a thread reaches a barrier, it performs a lock, decrements the `num-barrier` and performs an unlock. If the value of `num-barrier` is zero, it sets a `barrier_flag` to true. The thread then spins on a `barrier_flag` in its local cache until the value of `barrier_flag` becomes true, at which point the thread can proceed. The Sequent Balance 21000 software provides a similar solution. This algorithm takes $\Omega(N)$ time units to execute. For large N , the critical section would

7 EFFICIENT IMPLEMENTATION OF SYNCHRONIZATION VARIABLES

become a bottleneck.

The implementation of locks described above allows for a very efficient implementation of barriers in software. Our algorithm is based on "the tournament algorithm" proposed in [13] and [14]. The code for the barrier procedure is shown below. The algorithm uses a binary tree to synchronize the processes.

```
#define NUM_LOCKS 10 /* NUMLOCKS = number-of-processes - 1 */
mutex Lock_Array[NUM_LOCKS];
int flag[NUM_LOCKS];
int turn[NUM_PROCS];

Init()
  for(i=0;i<NUM_PROC;i++){
    Lock_Array[i]=i
    turn[i]=0;
  }
  for(i=0;i<NUM_LOCKS;i++){
    flag[i]=0;
  }
}

Barrier(process_id)
  index = process_id; /* from -6 to 0 for example */
  turn[index] = 1 - turn[index]
  while(index < NUM_LOCKS) {
    index = next[index];
    lock[Lock_Array[index]];
    Lock_Array[index]++;
    if (Lock_Array[index] == index) {
      unlock[Lock_Array[index]];
      while (flag[index] != turn[index]);
      left_child_index = left_child[index];
      right_child_index = right_child[index];
      flag[left_child_index] = 1 - flag[left_child_index];
      flag[right_child_index] = 1 - flag[right_child_index];
    }
    else {
      unlock[Lock_Array[index]];
      index--;
    }
  }
```

8 CONCLUSIONS AND FUTURE WORK

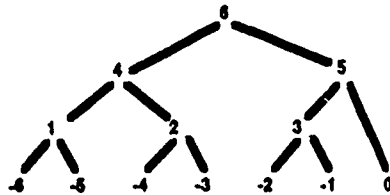


Figure 9: Static tree used for barrier synchronization

i	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5
next[i]	1	1	2	2	3	3	5	4	4	5	6	6

Table 6: Table representing the topology of the static tree

```

}
flag[index] = 1 - flag[index];

```

The basic idea of the algorithm is simple. A static binary tree of is used to synchronize pairs (*buddies*) of processes. The structure of the tree for a seven process barrier is shown in figure 9. Table 6 shows the table for the arrays used in the algorithm. At each level, the buddy that executes the lock successfully first, waits at that level. The *buddy* process that obtains the lock next goes to the next level in the tree. When two buddies reach the root of the tree, it implies that all processes have reached the barrier. This information is passed down the tree to the buddies that executed the lock first. The network traffic generated by our algorithm is $O(\log(N))$. Due to the efficient implementation of locks provided by the distributed directory cache coherence protocol, the software solution for barrier synchronization would produce minimal network traffic.

8 Conclusions and Future Work

We have presented a new protocol for providing cache coherence in large scale shared memory machines. The protocol provides efficient means to implement locks at minimal cost. The scalability and cost benefits of the protocol makes it appear to be a possible solution for the cache coherence problem in large scale shared memory multiprocessors. An analytic or

REFERENCES

simulation analysis of the protocol needs to be done to determine the performance. We will report the results of our performance analysis in a future paper.

References

- [1] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 281-289, 1988.
- [2] Thomas Anderson. The performance implications of spin-waiting alternatives for shared memory multiprocessors. In *1989 International Conference on Parallel Processing*, pages 170-174, 1989. Vol II.
- [3] James Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):274-298, November 1986.
- [4] James K. Archibald. *The Cache Coherence Problem in Shared Memory Multiprocessors*. PhD thesis, Department of Computer Science, University of Washington, February 1987. Available as Technical Report 87-02-06.
- [5] Philip Bitar and Alvin Despain. Multiprocessor cache synchronization. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 424-433, 1986.
- [6] Greg Byrd. Personal communication regarding talk given by Tom Knight at a workshop on ultra large scale message passing computers, 1987.
- [7] Lucien M. Censier and Paul Feautrier. A new solution to coherence problems in multi-cache systems. *IEEE Transactions on Computers*, c-27(12):1112-1118, December 1978.
- [8] W.J. Dally and C.L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5):547-553, May 1987.
- [9] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocesors. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 434-442, 1986.
- [10] D. Gelernter. A dag-based algorithm for prevention of store-and-forward deadlock in packet networks. *IEEE Transactions on Computers*, C-30:512-524, October 1981.

REFERENCES

- [11] James Goodman, Mary Vernon, and Philip Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64-75, April 1989.
- [12] K.D. Gunther. Prevention of deadlocks in packet-switched data transport systems. *IEEE Transactions on Communication*, COM-29:512-524, April 1981.
- [13] D. Hensen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1), 1988.
- [14] Boris Lubachevsky. Synchronization barrier and related tools for shared memory parallel programming. In *1989 International Conference on Parallel Programming*, pages 175-179, 1989. Vol II.
- [15] G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, A. Norton, and J. Weiss. The IBM Research Parallel Processor prototype (RP3): Introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764-771, 1985.
- [16] Richard Simoni. Implementing a directory-based cache consistency protocol. Unpublished report, Center for Integrated Systems, Stanford University, Stanford CA 94305., July 1988.

Distributed Cache Coherence for Large Scale Shared Memory Multiprocessors

Manu Thapar and Bruce Delagi

**Department of Electrical Engineering
Stanford University
Stanford, CA 94305**

and

**High Performance Systems
Digital Equipment Corporation
Palo Alto, CA 94301**

To appear in:

Proceedings of the 1989 International Conference on Intelligent Distributed Systems

Distributed Cache Coherence for Large Scale Shared Memory Multiprocessors*

Manu Thapar and Bruce Delagi

December 7, 1989

Abstract

This paper describes a new hardware solution and protocol for the cache coherence problem in large scale shared memory multiprocessors. The protocol is based on a linked list of caches and (to ensure a scalable design) does not require a global broadcast mechanism. Fully-mapped directory-based solutions proposed earlier also do not require a global broadcast mechanism. However, our solution has lower cost and potentially better performance than the fully-mapped directory-based protocol for high performance systems. Further, we do not assume that the network preserves the order of messages. Thus we do not preclude adaptive routing. Our solution also allows an efficient implementation of locks.

Key Words: Parallel systems, shared memory, cache coherence protocols.

1 Introduction

Cache coherence is an important well known problem in shared memory multiprocessor systems. If multiple caches are allowed to simultaneously have copies of a given memory location, a mechanism must exist to ensure that all copies remain consistent when the contents of that memory location are modified. Cache coherence protocols are well understood for bus-based shared memory architectures[3]. These protocols are also called snoopy cache coherence protocols. The term 'snoopy' is derived from the fact that each cache must watch all traffic on the bus and take appropriate action for addresses that are present in the cache. Addresses are, in effect, transmitted to each cache by global broadcast. The shared bus limits the number of processors to the number that can be connected to the bus without saturating it. To support scalable shared memory architectures, the cache coherence protocol needs to be able to work in the absence of a global broadcast mechanism. Centralized directory based schemes[1,7] are a possible solution in this environment. We present some drawbacks of these schemes in section 2.

Based on a linked list of caches[6], this paper describes a new distributed directory cache coherence protocol. The information about caches which have copies of the data is decentralized

*This work was supported by equipment provided by the Knowledge Systems Laboratory (KSL), Department of Computer Science, Stanford University.

and distributed among the cache lines. Our implementation, like the fully mapped centralized directory scheme [7], tracks any number of cache copies and never requires invalidates to be sent to all caches in the system. It has a lower cost and better performance than the fully mapped directory based coherence scheme for expected memory and cache sizes in high performance systems. In the fully mapped scheme, the size of the memory required to hold the state information is $O(MN)$, where M is the size of main memory and N is the number of caches. In our scheme, on the other hand, the size of the memory required to hold the state information is only $O(M \log N)$. We allow adaptive routing (so that network performance may be more robust) and thus do not assume that the interconnection network preserves the order of messages. The network traffic generated for invalidations is reduced by a factor of two in our implementation compared to a fully mapped directory scheme for networks that do not preserve the order of messages. An important feature of this protocol is that locks can be supported very efficiently with minimal extra cost.

This paper is organized as follows. Section 2 describes the directory based protocols and provides the necessary background for the rest of the paper. Section 3 describes the distributed directory protocol along with a simple performance analysis. Section 4 describes the implementation of locks. Section 5 states our conclusions and future work.

2 What's Wrong With Centralized Directory Entries?

In the fully mapped centralized directory scheme [7], the directory has N valid (or 'present') bits per line, where N is the number of caches. The amount of storage needed for the directory in the fully mapped scheme is thus $O(MN)$, where M is the size of main memory. If a cache has a copy of the line, the present bit corresponding to that cache is set. The directory also has a dirty bit. If the dirty bit is set, only one of the caches can have a copy of the line.

On a read miss, the directory is checked to see if the block is dirty in another cache. If so, consistency is maintained by copying the dirty block back to the memory before supplying the data. To ensure correct operation, the memory line has to be 'locked' by the directory controller until the write-back signal is received from the cache with the dirty block. No other coherence related operations on this line may be undertaken while a line is locked. If the line is not dirty in another cache, then data is supplied from the main memory and the corresponding present bit is set in the directory. On a write miss, the central directory is checked to determine the state of the line. If the line is dirty in another cache, then the line is first flushed from the cache before supplying the data. The memory line is locked while this is being done. If the line is clean in other caches, invalidate signals are sent to the caches. The memory line is locked until acknowledgements are received from the caches. The data can then be supplied to the requesting cache. Care has to be taken to handle race conditions properly. These race conditions and their handling are discussed in [4,14].

The locking of lines by the directory controller impacts the performance and design complexity of the cache coherence scheme. Requests that arrive while a line is locked have to be either buffered at the directory, or else bounced back to the source to be reissued at a later time. If the requests are buffered at the directory, the network traffic is lower. However, if the buffer overflows, the requests

still have to be bounced back. This complicates the design of the protocol (and the directory controller that implements it). If the directory controller is multithreaded, requests for a line that is not locked can be serviced immediately. However, implementing a multithreaded directory controller may be impractical due to its complicated design [14]. If the directory controller is not multithreaded, requests for no line can be serviced until the current request for that memory module is serviced. Locking the whole memory module while servicing a request for any line that requires a coherency related transaction could make the directory a potential bottleneck.

To reduce the amount of storage required, a number of modifications to the above scheme may be made. However, these modifications either require the implementation of an efficient broadcast mechanism (contradicting our assumption about scalable systems), or may generate excess network traffic along with performance penalties. For example, one simple modification is to have i pointers per line in the directory. Each pointer may point to a cache that has a copy of the line. If more than i caches have copies of the line, a broadcast has to be done to *all* caches to service a write miss. The memory line has to be locked until all caches acknowledge the invalidation. This is classified as a $Dir;B$ scheme in [1], where i is the number of indices kept in the directory and B stands for broadcast. A $Dir;NB$ scheme, where i is less than the number of caches and NB stands for no broadcast, is possible also. In such a scheme, at most i caches can have copies of a line at the same time. In the case where a read miss occurs when i caches have copies of the line, the directory has to invalidate one of the copies before the data can be supplied to the requesting cache. This might result in "thrashing" the line between caches.

3 How Would Distributed Directory Entries Be Better?

We will assume a very general computing system structure in our discussions. Figure 1 describes this basic architecture. Each node consists of one or more processing elements (P), a cache (C), a network controller (NC) and part of the distributed global memory (DGM). The DGM includes the directory. For the distributed directory protocol we do not assume that the network preserves the order of messages. This allows adaptive routing (making network performance more robust). We are not aware of directory based protocols described in the literature that allow out of order message arrival.

3.1 The Cost is Lower

In our distributed directory protocol, caches that share data are linked together in a list. We first present the general outline of the protocol here and explain it in further detail in section 3.2. Each line in the main memory and the cache has a cache-pointer field associated with it. This pointer can address any cache in the system. The directory services a read or write miss request by changing the cache-pointer in the directory entry associated with the line to point to the requesting cache. If the old value of the cache-pointer is nil, a reply is sent directly to the requesting cache. If the old value of the cache-pointer points to a cache, the request is forwarded to that cache. In case of read misses, that cache replies to the requesting cache, and the distributed list now includes the requesting cache. In case of write misses, the distributed list has to be invalidated before a reply

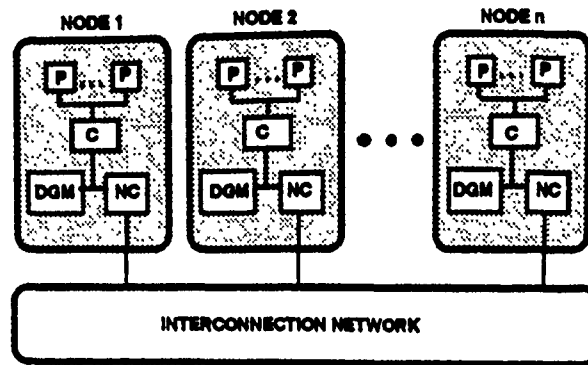


Figure 1: The basic architecture

can be sent to the requesting cache. A read miss reply consists of the data and address of the cache or memory module replying to the request. A write miss reply consists of the data.

The amount of memory required for the pointer is $\log N$ where N is the number of caches. The total amount of memory needed is thus $O(M \log N + Nc \log N)$ where M is the total size of main memory, N is the number of caches and c is the size of each cache. The above expression can be written as $O(M \log N(1 + k))$ where k is Nc/Nm (m being the amount of memory per node). We interpret k as the ratio of the size of cache memory per node to the size of main memory per node.

Assuming a constant value of k for the machine, the amount of memory required for the distributed directory scheme is $O(M \log N)$. We can expect then, that the cost of implementing the distributed directory scheme is significantly less than the fully mapped scheme—which requires $O(MN)$ amount of memory.

3.2 The Distributed Directory Controller Need Not Be Multithreaded

In the distributed directory protocol, the directory controller is not multithreaded. The servicing of requests does not require any locking of lines as in the case of the centralized directory protocol. In case of cache misses, the cache controller issues requests to the appropriate memory module. A reply is subsequently received, either from the main memory, or from another cache. The communication between the caches and the main memory may be considered to be similar to asynchronous message passing. The cache controller sends a miss signal to the main memory on read and write misses. Before a miss signal is issued by a cache controller, a line associated with the address is allocated in the cache memory. All signals specify the address of the source of the request and the address of the lines they are associated with.

A line in main memory is originally in state 'absent' from all caches. Each request causes the value of the cache-pointer to be updated to point to the requesting cache. If the line is absent from all the caches, the main memory sends a reply. Otherwise the request is forwarded to the last cache to make a request for the same line.

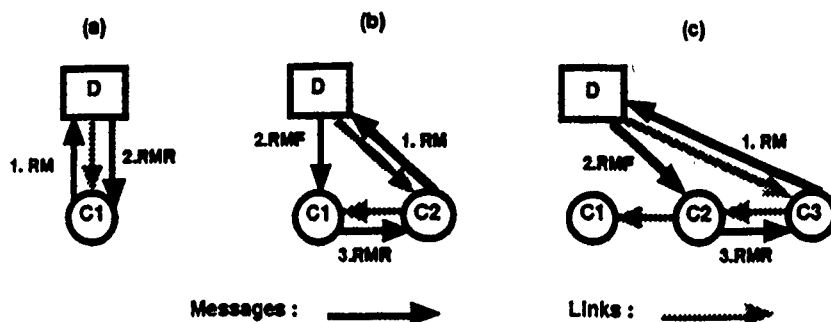


Figure 2: Linking of caches due to read misses

A line in cache memory is originally in state 'invalid'. A read or a write request from the processor causes the state to change to 'writing-or-reading' and a read-miss or write-miss signal to be sent to the appropriate main memory module. On a read-miss-reply, the value of the cache-pointer is set to be the address of the object sending the reply. This causes a linked-list of caches that contain the data in shared state to be formed. Figure 2 illustrates the process followed to set up the linked list. Consider the case where cache C1 has a read miss for a line followed by caches C2 and C3. As shown in fig. 2(a), cache C1 sends a read-miss signal to the directory. The cache-pointer of the line in the directory is made to point to C1. Since no other cache has a copy of the line, the main memory sends a read-miss-reply to C1. When C1 receives the reply the line is loaded into the cache in state 'shared'. Now, when cache C2 sends a read-miss to the directory, a read-miss-forward signal is sent to C1 as shown in fig. 2(b). The directory does not send a reply directly to C2 since C1 may have written to the line locally. The cache-pointer in the directory now points to C2. When C1 receives the forwarded signal, it sends a reply to C2. The reply includes the data and the address of C1. When C2 receives the reply, it sets its cache-pointer to point to C1. Thus a linked list is formed. Fig. 2(c) shows how C3 gets linked into the list.

Write misses cause a write-miss signal to be sent to the directory. As mentioned earlier, a line is allocated in the cache before the miss signal is sent. This line is used to buffer the write. Write buffering[8,13] allows the processor to proceed immediately without stalling. A write is considered to be *issued* when a write-miss is sent by the cache. A write is considered to be *performed* when a write-miss-reply is received by the cache. A *fence* [12] operation may be used to ensure that all writes that have been issued by a processor are performed before that processor is allowed to proceed. If a copy of the line is not present in any other cache, the main memory directly sends a reply. Otherwise the copies of the line have to be invalidated before a reply can be sent. Figure 3 shows the sequence of events that result when multiple caches have a copy of the line and C4 has a cache miss. The directory forwards the write miss signal to the old head pointed to by the cache-pointer and the cache-pointer is updated to point to C4. When C3 receives the write-miss-forward signal, it invalidates its copy and forwards the signal to C2. C2 does the same and forwards the signal to C1. Since the cache-pointer of C1 points to the directory, it can be determined locally

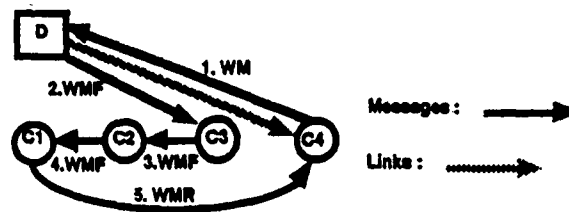


Figure 3: Invalidations due to write misses

that C1 is the tail of the list and a reply is sent to C4 after the data in C1 is invalidated.

If the line in the cache is in state 'writing-or-reading' and a read-miss-forward or a write-miss-forward signal is received, the forwarded signal is stored in the cache-pointer field of the cache line. The state is changed to note that a forwarded signal has been stored. Such signals are serviced when the reply to the local read or write miss is received. If multiple transactions for the same line are pending, the caches form a distributed queue of them. Details of the protocol may be found in [15].

As we have seen, when the directory controller receives a write or read miss request, and the data is present in some cache(s), the request is forwarded to the head of the list. Neither the memory line nor the memory module needs to be locked. The directory controller is single threaded and we do not have the problem of buffering the signals and bouncing them to the sources as did the centralized directory protocols mentioned in section 2. No locking occurs during coherence related transactions and replies to requests are sent by caches directly to the requesting cache. This helps to prevent the directory from becoming a potential bottleneck.

3.3 The Protocol Has Good Performance

In this section we shall present some qualitative discussion and simple quantitative analysis to show that besides saving in cost, the distributed directory protocol has good performance. As mentioned in section 2, modified versions of the fully mapped centralized directory protocol may be used to provide a saving in cost. In the *Dir, B* protocol, memory space for i pointers per memory line has to be provided for the directory. When the number of readers exceeds i , a broadcast has to be done to invalidate the copies on a write miss. Applications that have more than i readers between successive writes would show poor performance on such a system since broadcasts would have to be done frequently. In a *Dir, NB* protocol, such applications would result in 'thrashing' between caches. The distributed directory protocol does not have any of these problems. As the number of readers between writes increases, the length of the linked list of readers increases and though there is some degradation in the performance, it is more graceful. We shall now compare the performance with the fully mapped centralized directory protocol.

Use of write-buffering, the performance of point-to-point signal transmission schemes and access

times for static cache memory (SRAM) and dynamic main memory (DRAM) are critical issues in contrasting centralized and distributed directory protocols. We shall first state the assumptions we make to compare the distributed directory protocol with the centralized directory protocol and then use a simple, commonly occurring problem, involving communication of each process with a given number of other processes, to do an approximate analysis. In our analysis we also show how some variations in our assumptions would affect relative performance.

Currently the ratio of access times for static cache memory (SRAM) and dynamic main memory (DRAM) for high performance SRAMs and DRAMs is around 1:10 [9,10]. Using surface mount packaging, we estimate point to point transmission times in the network to be equal to the cycle time of the SRAM. We also assume that the directory is implemented in SRAM. In a 256 node machine, the average distance that a message has to travel requires approximately 10 cycles. This we take as the time for a "network operation".

In the centralized directory scheme, a read miss would require 2 network operations (to the main memory and back) and a DRAM access (for the main memory) if the line is not dirty in another cache. For a SRAM to DRAM access time ratio of 1:10, this would imply a total latency of about 32 cycles (20 for the network operations, 10 for the DRAM access and 2 for the SRAM access). If the line is dirty in another cache, 2 extra network operations would be required to get the data from that cache. This would imply a total latency of about 52 cycles. In the distributed directory scheme, if the data is present in another cache, no DRAM accesses are required since the data is obtained from SRAM cache. A total of 3 network operations are required. This implies a latency of about 32 cycles (whether the line is dirty or not). Due to the point to point network performance, direct cache to cache transfers, and faster access times for SRAM memory, read misses would result in less (or comparable) latency in the distributed directory scheme.

The latency of write misses may be a possible cause of concern since the invalidations of the caches linked in the list have to be done sequentially. However, if the writes to a line occur frequently, the number of caches that have to be invalidated between writes will be small. Thus the cases when the latency is large will be infrequent. Measurements on a range of applications[16] supports the assertion that the number of caches that have to be invalidated on writes is (on average) small.

Additionally, we use write buffering to reduce the effect of sequential invalidation operations. Write buffering is frequently used in uniprocessor systems to prevent the processor from stalling on a write miss. The writes to memory are buffered in the cache and the processor is allowed to proceed. In multiprocessor systems, write buffering has to be done with care to avoid unexpected consistency violations. The problem of write buffering in multiprocessors has been studied in considerable detail[8,13]. In the distributed directory protocol a write is performed when the write-miss-reply is received by the requesting cache. Before exiting a critical section, a fence is executed to ensure that all writes are performed.

We now consider an analysis of the distributed directory protocol in the context of an algorithm. Many algorithms require communication with a fixed number of processes that are operating on data elements. In such algorithms a process associated with an element usually reads the values of the logical neighbors of the element, computes a value and then issues a write instruction. We call the number of logical neighboring elements that read the updated value to be the *fan-out* of the process. The fan-out may be considered to be the number of readers between successive writes

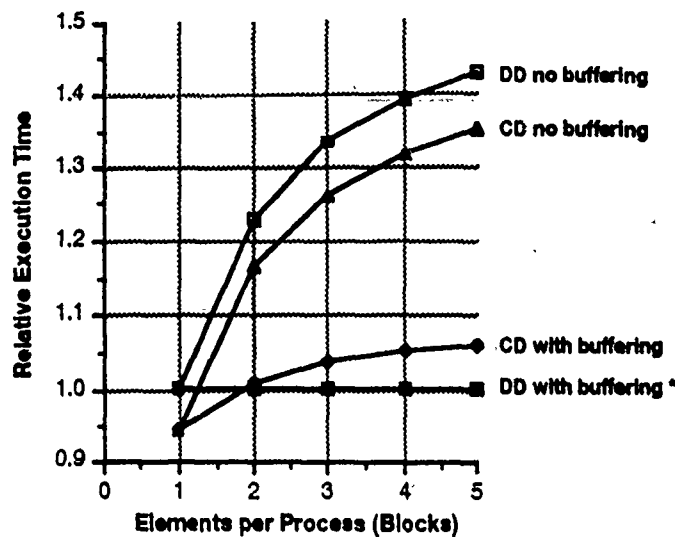


Figure 4: Effect of write buffering (fan-out = 4; SRAM/DRAM = 1:10; network operation = 10 cycles)

to a variable. Consider the sequence of reads and a write to constitute a *block*. In the case where there is more than one element per process, it may be possible to operate on several blocks and so buffer the writes of several blocks before having to perform a fence operation. Figure 4 shows the relative execution time versus the number of elements per process (the number of blocks) that can be operated upon before a fence operation is required. The execution time was calculated by analyzing the number of network operations required for each scheme and adding up the time needed to do these operations. The base case is shown by a '*'. *

For figure 4 we have assumed a fan-out of 4, the SRAM to DRAM access time ratio to be 1:10 and the number of cycles required for a network operation to be 10. Write buffering improves the performance of both the centralized and the distributed directory protocols. However, the improvement in performance for the distributed directory protocol is more than the improvement in performance for the centralized directory protocol.

Figure 5 shows the effect on performance with a change of SRAM to DRAM access time ratio. In this figure we have assumed write buffering for all cases. In the distributed directory protocol, the data is transferred between caches most of the time. Therefore, the performance of the distributed directory protocol would not vary much with a change in DRAM access time. A lower DRAM access time favors the centralized directory protocol. The distributed directory protocol allows the use of cheaper, slower, and larger DRAM memory without causing a noticeable loss in performance.

Figure 6 shows the effect of changing the number of cycles required for a network operation. A lower number of cycles for a network operation favors the distributed directory protocol more than the centralized directory protocol.

Figure 7 shows the effect of varying the fan-out of the algorithm. A larger fan-out implies that more caches have to be invalidated on a write. For example, with a fan-out of 16, 16 caches have to be invalidated on each write. The use of write buffering and the lower latency for reads helps

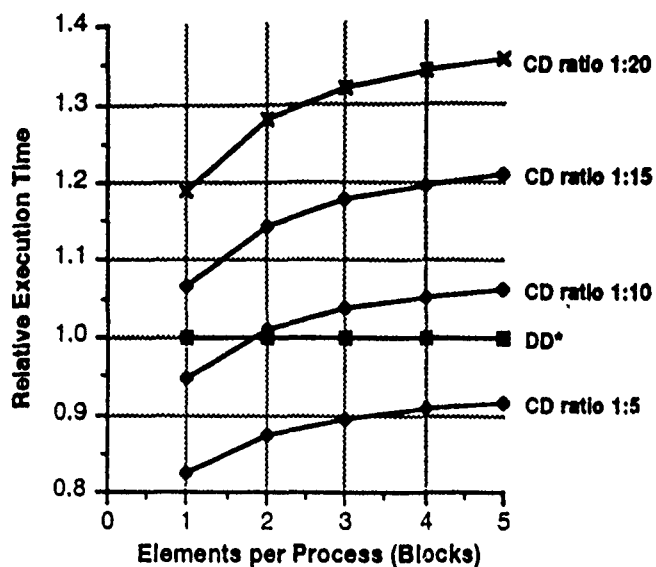


Figure 5: Varying SRAM/DRAM ratio (fan-out = 4; network operation = 10 cycles)

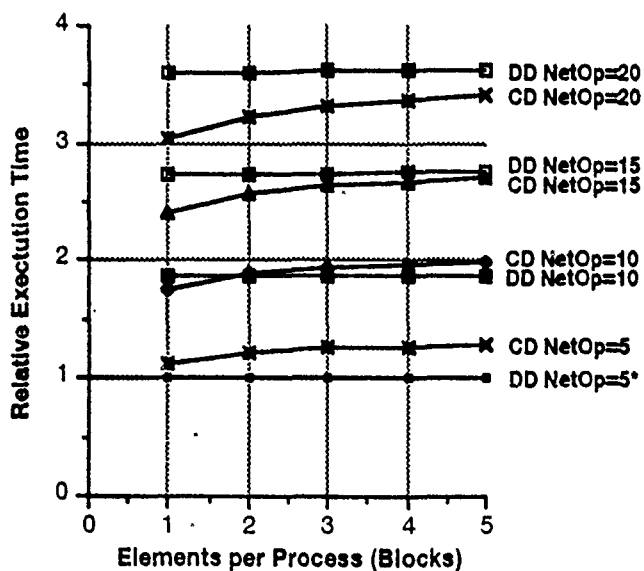


Figure 6: Varying the cycles for a network operation (fan-out = 4; SRAM/DRAM = 1:10)

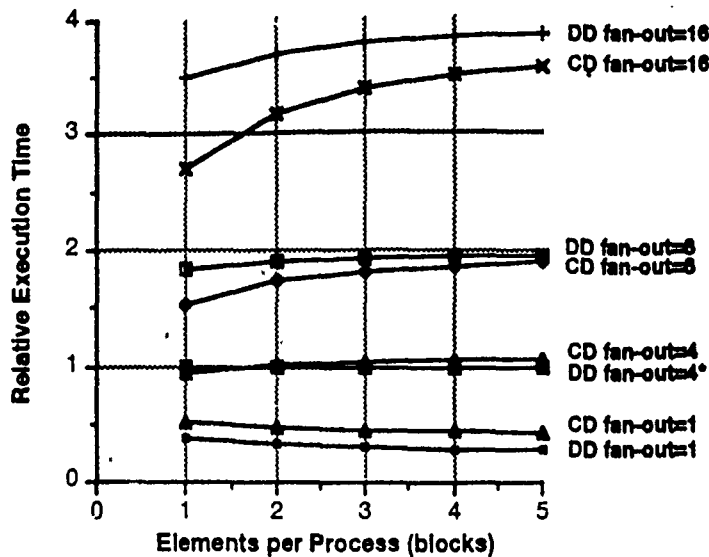


Figure 7: Varying the fan-out (SRAM/DRAM = 1:10; network operation = 10 cycles)

to reduce the effect of sequential invalidations in the distributed directory protocol. It should be noted here that our analysis does not consider network or resource contention. In the centralized directory scheme, there would be more contention for the resources when the fan-out increases. This should worsen the performance curves for the centralized directory scheme. The distributed directory protocol should perform better when the fan-out is less than 4 and have comparable performance when the fan-out is larger.

A more accurate analysis would need detailed simulations. We are in the process of constructing a simulation model of the protocol. However, our simple analysis shows that besides savings in cost (as discussed in section 3), the distributed directory protocol can provide good performance as well.

4 Efficient Implementation of Synchronization Variables

In shared memory architectures it is very important to handle locks and barriers efficiently. The distributed cache coherence scheme can be used to implement a very efficient scheme of locks at minimal extra cost.

Familiar microprocessor architectures have some form of atomic test&set instruction to implement spin locks[2]. The test&set instruction sets the value of a memory location and atomically returns the old value. When a process wants access to a lock, the processor performs the test&set instruction and if the operation is successful, the processor continues. Otherwise the processor repeatedly tries to access the lock until it is successful. Spinning on a test&set instruction can cause a lot of network traffic each time a lock is released. A better alternative is a test&test&set instruction, where the first test is done in the cache, and the test&set done only if the first test is successful. This will reduce the network traffic, but even if this is done, the network traffic due

to lock acquisition is $O(N^2)$ where N equals the number of processors that are contending for the lock¹. Further, such implementations of locks can result in starvation because the accessing of locks is not fair (perhaps due to differences in the network distance between a lock and its contenders).

The distributed cache coherence protocol allows an implementation of locks that makes the network traffic minimal. Lock requests are queued and normally serviced in FIFO order. The network traffic is only $O(N)$.

Fine grain locking is provided by having a lock bit per line[5]. The main advantage of providing a lock bit per line is that the data can be obtained at the same time as the lock. The code for a lock procedure is shown below.

```
procedure lock(var)
begin
    Queue-Lock(var)
    while (Test&Set(var))
        Queue-Lock(var)
end
```

The queue-lock instruction is used to join a queue of nodes waiting for a lock. A cache line with the lock bit set is allocated on a queue-lock instruction. A queue-lock instruction may be issued more than once. If the associated line has already been allocated in the cache, the instruction is redundant and has no effect. The test&set instruction returns a value of '1' until the lock is obtained at the local cache.

The queue of locks is formed in a similar way as the linked list for caches that have shared copies of a line. The implementation of locks requires a few extra states and signals. When a queue-lock instruction is executed, a line in the state 'lock-in-progress' is allocated and a lock-miss signal is sent to the directory. If no other cache has locked the line, the directory sends a lock-granted signal along with the data². Otherwise, the directory updates its cache-pointer to point to the requesting cache and forwards the lock-miss signal to the cache previously pointed to by cache-pointer. The cache that receives this forwarded signal, stores the address of the requesting cache (the *downstream* pointer) in its cache-pointer field. A set-upstream-pointer signal with the old value of the cache-pointer is also sent to the requesting cache³. The requesting cache stores this *upstream* pointer in its data field. In this way, a doubly linked list of caches waiting for a lock is formed. Lines that are in state 'lock-in-progress' do not have valid data, so these lines can store the pointers for the doubly linked list without requiring extra memory storage. The 'lock-in-progress' state has a few *sister states* that are used to ensure that the signals required to form the doubly linked list are received.

More than one processor may share a cache and there may be more than one process per processor contending for the same lock. When there are multiple processes per processor contending for the same lock, the lock procedure is executed by each process. The first queue-lock request to reach the cache causes a line in the cache to be linked into the doubly linked list of caches waiting

¹To check this, note that each time a lock is released, all contenders for the lock try to access it.

²The first cache that receives the lock-granted signal from the directory is considered as the upstream end.

³The last requesting cache is considered as the downstream end. The directory points to the downstream end.

for a lock. The processes associated with a cache contend for the lock by checking the cache locally, without generating any network traffic. When a lock-granted signal is received, a process has to obtain the lock and release it to allow other caches waiting for the lock to get the lock. If no other cache wants the lock, the line is held locally. Locking and unlocking of the line by processes that share the cache can now be done in the cache very efficiently without requiring any network traffic. After the lock has been used and released once locally, and if a lock-miss-forward signal is received, the lock is granted to the next downstream cache in the queue. This ensures fairness between processors. In our distributed directory scheme, the grant signal does not have to go through the directory and the data is passed between the caches.

Unlocking of locked lines by processors other than the one that performed the lock is allowed. An unlock signal is sent to the directory which then forwards the signal to the downstream end of the queue of caches. The signal flows upstream until it reaches the node with the locked line. This facility is provided to allow migration of processes between processors.

The advantages of this scheme are that except for replacements, lock requests are serviced in FIFO order and starvation is eliminated if there is one process per cache and this solution requires only $O(N)$ operations⁴. Details of the implementation of locks including handling of replacement for the distributed directory scheme may be found in [15].

5 Conclusions and Future Work

The work presented in this paper represents a first step at evaluating a distributed directory protocol. To obtain more accurate results, simulations need to be done and so we are in the process of developing a simulation model of the protocol. The distributed directory protocol presented in this paper is an invalidate based protocol but update based and hybrid protocols are possible to implement also. Hybrid protocols could update a part of the list and invalidate the rest. It would be useful to determine the tradeoffs of such variations.

We have presented a new protocol for providing cache coherence in large scale shared memory machines. A simple performance analysis of the protocol has given encouraging results. The implementation of the protocol provides an efficient implementation of locks at minimal cost. The scalability and cost benefits of the implementation provides us with enough reasons to conclude that the distributed directory protocol may be considered to be a possible solution for the cache coherence problem in large scale shared memory multiprocessors.

Acknowledgements: We are thankful to Greg Byrd, Mike Flynn and Max Hailperin for their comments and suggestions and to the members of the DEC High Performance Systems Group and KSL for their support.

⁴A somewhat similar scheme has been proposed by Goodman *et.al.*[11]. However, their implementation requires the interconnect to support global broadcasts and the transfer of the lock and data to the next cache in the queue requires interactions with the main memory, resulting in more network traffic.

References

- [1] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 281-289, 1988.
- [2] Thomas Anderson. The performance implications of spin-waiting alternatives for shared memory multiprocessors. In *1989 International Conference on Parallel Processing*, pages 170-174, 1989. Vol II.
- [3] James Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):274-298, 1986.
- [4] James K. Archibald. The cache coherence problem in shared-memory multiprocessors. Ph.D. Thesis available as Technical Report 87-02-06, Department of Computer Science, University of Washington, February 1987.
- [5] Philip Bitar and Alvin Despain. Multiprocessor cache synchronization. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 424-433, 1986.
- [6] Greg Byrd. Personal communication regarding comments made by Tom Knight at the workshop on ultra large scale message passing computers, 1987.
- [7] Lucien M. Censier and Paul Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, c-27(12):1112-1118, December 1978.
- [8] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 434-442, 1986.
- [9] M. Odaka et.al. A 512Kb/5ns BiCMOS RAM with 1KG/150ps logic gate array. In *1989 IEEE International Solid-State Circuits Conference*, pages 28-29, 1989.
- [10] T. Takeshima et.al. A 55ns 16Mb DRAM. In *1989 IEEE International Solid-State Circuits Conference*, pages 246-247, 1989.
- [11] James Goodman, Mary Vernon, and Philip Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64-75, April 1989.
- [12] G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, A. Norton, and J. Weiss. The IBM Research Parallel Processor prototype (rp3): Introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764-771, 1985.

- [13] Christoph Scheurich and Michel Dubois. Correct memory operation of cache-based multiprocessors. In *Proceedings of the 14th International Symposium on Computer Architecture*, pages 234-243, 1987.
- [14] Richard Simoni. Implementing a directory-based cache consistency protocol. Unpublished report, Center for Integrated Systems, Stanford University, Stanford CA 94305., July 1988.
- [15] Manu Thapar and Bruce Delagi. Design and implementation of a distributed directory cache coherence protocol. KSL 89-72, Stanford University, 1989.
- [16] Wolf-Dietrich Weber and Anoop Gupta. Analysis of cache invalidation patterns in multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243-256, 1989.

Cache Coherence for Large Scale Shared Memory Multiprocessors

Manu Thapar and Bruce Delagi

Department of Electrical Engineering
Stanford University
Stanford, CA 94305

and

Highly Parallel Systems and Applications
Digital Equipment Corporation
Palo Alto, CA 94301

To appear in:

Proceedings of the 1990 ACM Symposium on Parallel Algorithms and Architectures

Cache Coherence for Large Scale Shared Memory Multiprocessors

Manu Thapar and Bruce Delagi
Digital Equipment Corporation
Stanford University

1 Introduction

This paper describes a new solution for the cache coherence problem in large scale shared memory multiprocessors. The protocol is based on a linked list of caches — forming a *distributed directory* and (to ensure a scalable design) does not require a global broadcast mechanism. Our solution has a lower cost and potentially better performance than current solutions that do not require a global broadcast. The performance of the protocol is more robust when there is contention for the data and for variations in memory technology. Further, we do not assume that the network preserves the order of messages. Thus we do not preclude adaptive routing. Our solution also allows an efficient implementation of locks.

Cache coherence is an important well known problem in shared memory multiprocessor systems. If multiple caches are allowed to simultaneously have copies of a given memory location, a mechanism must exist to ensure that all copies remain consistent when the contents of that memory location are modified. Cache coherence protocols are well understood for *bus-based* shared memory architectures [2]. These protocols, called "snoopy" cache coherence protocols, require that each cache watch all traffic on the bus and take appropriate action for addresses that are present in that cache. Addresses are, in effect, transmitted to each cache by global broadcast. The shared bus limits the number of processors to the number that can be connected to the bus without saturating it. To support *scalable* shared memory architectures, the cache coherence protocol needs to be able to work in the absence of such a global broadcast mechanism (bus-based or

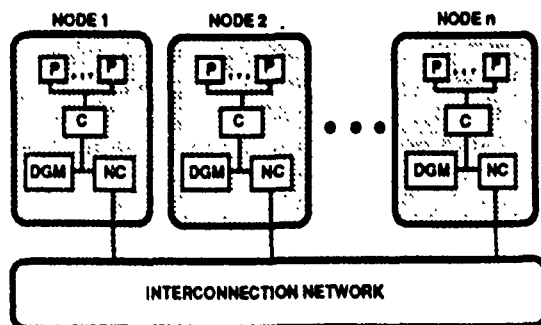


Figure 1: The basic architecture

otherwise). Centralized directory based schemes [1, 6] are a possible solution in this environment.

We present some drawbacks of these schemes in section 2 and contrast them to an alternative, distributed directory protocol, described in section 3. Section 4 states our conclusions and future work.

2 What's Wrong With Centralized Directory Entries?

We will assume a very general computing system structure in our discussions. Figure 1 describes this basic architecture. Each node consists of one or more processing elements (P), a cache (C), a network controller (NC) and part of the distributed global memory (DGM). The DGM includes the directory.

In the directory based protocols there is a directory "tag" associated with each line in main memory. This directory is used to hold information about which caches have copies of the line. In the fully mapped centralized directory scheme [6], the directory has N valid (or "present") bits per line, where N is the number of caches. The amount of storage needed for the directory

in the fully mapped scheme is thus $O(MN)$, where M is the size of main memory. If a cache has a copy of the line, the present bit corresponding to that cache is set. The directory also has a dirty bit. If the dirty bit is set, only one of the caches can have a copy of the line.

On a read miss, the directory is checked to see if the block is dirty in another cache. If so, consistency is maintained by copying the dirty block back to the memory before supplying the data. To ensure correct operation, the memory line has to be "locked" by the directory controller until the write-back signal is received from the cache with the dirty block. No other coherency-related operations on this line may be undertaken while a line is locked. If the line is not dirty in another cache, then data is supplied from the main memory and the corresponding present bit is set in the directory. On a write miss, the central directory is checked to determine the state of the line. If the line is dirty in another cache, then the line is first flushed from the cache before supplying the data. The memory line is locked while this is being done. If the line is clean in other caches, invalidate signals are sent to the caches. The memory line is locked until acknowledgements are received from the caches. The data can then be supplied to the requesting cache.

The locking of lines by the directory controller impacts the performance and design complexity of the cache coherence scheme. Requests that arrive while a line is locked have to be either buffered at the directory, or else bounced back to the source to be reissued at a later time. If the requests are buffered at the directory, the network traffic is lower. However, if the buffer overflows, the requests still have to be bounced back. This complicates the design of the protocol (and the directory controller that implements it). Locking of the lines while servicing a request that requires a coherency-related transaction could make the directory a bottleneck.

To reduce the amount of storage required, a number of modifications to the above scheme may be made. However, these modifications either require the implementation of an efficient broadcast mechanism (contradicting our assumption about scalable systems), or may generate excess network traffic along with performance penalties. For example, one simple modification is to have i pointers per line in the directory. Each pointer may point to a cache that has a copy of the line. If more than i caches have copies of the line, a broadcast has to be done to all caches to service a write miss. The memory line has to be locked until all caches acknowledge the invalidation. This is classified as a Dir_iB scheme [1], where i is the number of indices kept in the directory and B stands for broadcast. A Dir_iNB

scheme, where i is less than the number of caches and NB stands for no broadcast, is possible also. In such a scheme, at most i caches can have copies of a line at the same time. In the case where a read miss occurs when i caches have copies of the line, the directory has to invalidate one of the copies before the data can be supplied to the requesting cache. This might result in "thrashing" the line between caches.

3 How Would Distributed Directory Entries Be Better?

Based on a linked list of caches [4], we now define our distributed directory protocol. For the distributed directory protocol we do not assume that the network preserves the order of messages. This allows adaptive routing (making network performance more robust). We are aware of no other directory based protocol previously described in the literature that allows out-of-order message arrival.

In our distributed directory protocol, caches that share data are linked together in a list. Each line in the main memory and the cache has a cache-pointer field associated with it. This pointer can address any cache in the system. The directory services a read or write miss request by changing the cache-pointer in the directory entry associated with the line to point to the requesting cache. If the old value of the cache-pointer is nil, a reply is sent directly to the requesting cache. If the old value of the cache-pointer points to a cache, the request is forwarded to that cache. In case of read misses, that cache replies to the requesting cache, and the distributed list now includes the requesting cache. In case of write misses, the distributed list has to be invalidated before a reply can be sent to the requesting cache.

3.1 The Cost is Lower

The amount of memory required for the cache-pointer is $\log N$ where N is the number of caches. The total amount of memory needed is thus $O(M \log N + Nc \log N)$ where M is the total size of main memory, N is the number of caches and c is the size of each cache. The above expression can be written as $O(M(1+k) \log N)$ where k is Nc/Nm (m being the amount of memory per node). We interpret k as the ratio of the size of cache memory per node to the size of main memory per node.

Assuming a constant value of k for the machine, the amount of memory required for the distributed directory scheme is $O(M \log N)$. We can expect then

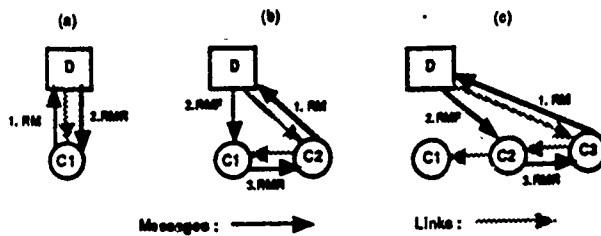


Figure 2: Linking of caches due to read misses

that, using the same technology, the cost of implementing the distributed directory scheme is significantly less than the fully mapped scheme—which requires $O(MN)$ amount of memory.

3.2 Resource Utilization is Distributed

In the distributed directory protocol, the information about which caches have copies of the data is distributed among the cache lines. The servicing of requests does not require any locking of lines as in the case of the centralized directory protocol. Direct cache-to-cache operations are used to send the replies and none of the replies have to be serialized through the main memory. The centralized bottleneck which is present in the centralized directory protocols is thus eliminated or significantly reduced.

A line in cache memory is originally in state “invalid”. A read or a write request from the processor causes the state to change to “writing-or-reading” and a read-miss or write-miss signal to be sent to the appropriate main memory module. On a read-miss-reply, the value of the cache-pointer is set to be the address of the cache sending the reply (the cache-pointer remains nil if the reply was sent by the main memory). This causes a linked list of caches that contain the data in shared state to be formed.

Figure 2 illustrates the process followed to set up the linked list. Consider the case where cache C1 has a read miss for a line followed by caches C2 and C3. As shown in fig. 2(a), cache C1 sends a read-miss signal to the directory. The cache-pointer of the line in the directory is made to point to C1. Since no other cache has a copy of the line, the main memory sends a read-miss-reply to C1. When C1 receives the reply, the line is loaded into the cache in state “exclusive”. Now, when cache C2 sends a read-miss to the directory, a read-miss-forward signal is sent to C1 as shown in fig. 2(b).

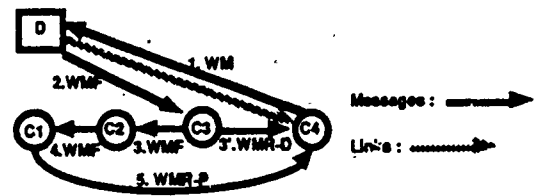


Figure 3: Invalidations due to write misses

The directory does not send a reply directly to C2 since C1 may have written to the line locally. The cache-pointer in the directory now points to C2. When C1 receives the forwarded signal, it sends a reply to C2 and changes its state to shared. The reply includes the data and the address of C1. When C2 receives the reply, it sets its cache-pointer to point to C1. Thus a linked list is formed. Fig. 2(c) shows how C3 gets linked into the list.

Write misses cause a write-miss signal to be sent to the directory. A line is allocated in the cache before the miss signal is sent. This line is used to buffer the write. Write buffering along with weak ordering [7, 9] allows the processor to proceed immediately without stalling. A write is considered to be *issued* when a write-miss is sent by the cache. A write is considered to be *performed* when a write-miss-reply is received by the cache. A write-miss-reply may consist of two signals as in the example below. A *fence* [8] operation may be used to ensure that all writes that have been issued by a processor are performed before that processor is allowed to proceed. If a copy of the line is not present in any other cache, the main memory directly sends a reply. Otherwise the copies of the line have to be invalidated before a reply can be sent.

Figure 3 shows the sequence of events that result when multiple caches have a copy of the line and C4 has a cache miss. The directory forwards the write miss signal to the old head (C3) pointed to by the cache-pointer and the cache-pointer is updated to point to C4. When C3 receives the write-miss-forward signal, it invalidates its copy and forwards the signal to C2. C3 also sends a write-miss-reply-data signal along with the requested data to the requesting cache C4. When C2 receives the write-miss-forward signal, it invalidates its copy and forwards the signal to C1. Since the cache-pointer of C1 points to the directory, it can be determined locally that C1 is the tail of the list and a write-miss-reply-performed signal is sent to C4 after the data in C1 is invalidated. C4 needs to receive both the write-miss-reply-data and the write-miss-reply-performed signals

before the write can be considered to be performed.

A cache line is in state "writing-or-reading" after a read-miss or a write-miss has been generated and before a read-miss-reply or a write-miss-reply has been received. If the line in the cache is in state "writing-or-reading" and a read-miss-forward or a write-miss-forward signal is received, the forwarded signal is stored locally. This is done by writing the address of the requesting cache in the cache-pointer field of the cache line and changing the state to note that a forwarded signal has been stored. Such signals that are stored are called *pending signals* and are serviced when the reply to the local read or write miss is received. If multiple transactions for the same line are pending, the caches form a *distributed queue* of pending signals. The requests are thus serviced in a pipelined manner rather than causing contention at the directory as in the case of the centralized directory protocol.

Replacement of lines that are part of a shared list is done by invalidating the lower portion of the list. If care is not taken, the forwarding of signals may lead to deadlocks in the network since most network routing protocols assume that a message is *consumed* at its destination. One way of taking care of potential deadlocks is to use more than one logical network. Details of the protocol including handling of potential races may be found in [10].

As we have seen, when the directory controller receives a write or read miss request, and the data is present in some cache(s), the request is forwarded to the head of the list. We do not have the problem of locking lines at the main memory, buffering the signals and bouncing them to the sources as did the centralized directory protocols (as mentioned in section 2). Requests are serviced in a more decentralized fashion and the replies do not have to be serialized through the directory as in the case of the centralized directory protocol. This helps to prevent the directory from becoming a potential bottleneck.

3.3 The Protocol Has Good Performance

The distributed directory protocol has good performance. The latency of write misses may be a possible cause of concern since the invalidations of the caches linked in the list have to be done sequentially. However, if the writes to a line occur frequently, the number of caches that have to be invalidated between writes will be small. Thus the cases when the latency is large will be infrequent. Measurements on a range of applications [12] support the assertion that the number of caches that have to be invalidated on writes is (on aver-

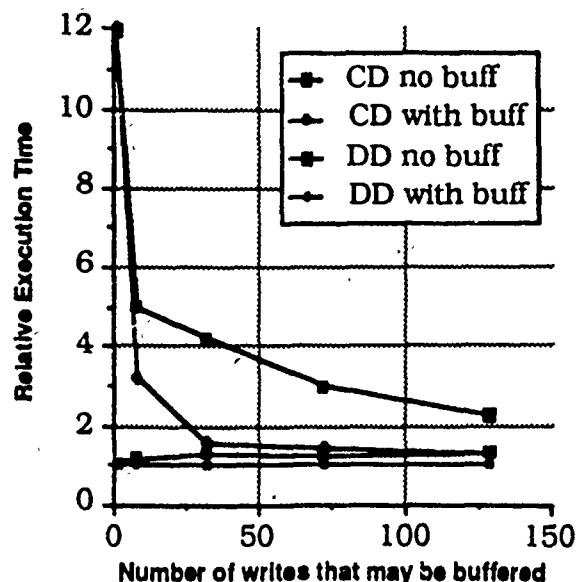


Figure 4: Performance comparison with and without write buffering

age) small. Additionally, we use write buffering along with weak ordering to reduce the effect of sequential invalidation operations. Write buffering is frequently used in uniprocessor systems to prevent the processor from stalling on a write miss. The writes to memory are buffered in the cache and the processor is allowed to proceed. In multiprocessor systems, write buffering has to be done with care to avoid unexpected consistency violations. The problem of write buffering in multiprocessors has been studied in considerable detail [7, 9].

We studied the performance of the fully mapped centralized directory protocol and the distributed directory protocol by using an explicit partial differential equation (PDE) solver as a benchmark. A PDE algorithm was chosen since they are widely used in scientific and engineering communities in applications requiring high performance computation [3]. Fully associative, infinite caches were assumed for the simulation. The data was uniformly distributed and the computational threads were scheduled at random sites so as not to favor the distributed directory protocol. Each node had one thread running on it. The same uniform distribution of the data and random placement of the threads was used for the comparisons. The simulation models were built upon an event driven simulation environment that has been used for other studies of multiprocessor operation [5]. A mesh topology with 32-bit bidirectional channels was used for the network. The memory line size was assumed to be 64 bytes and both the cache and the main memory were assumed to be 32

bits wide. The SRAM cache to DRAM main memory access ratio was assumed to be 1:10. The directories for both the protocols was assumed to be implemented in SRAM whose cycle time was taken to be 1 cycle. The network was assumed to transfer data between neighboring nodes in 1 cycle. The simulation results presented here were obtained by using 16-processor models. Preliminary results obtained for larger models show that our results should scale for larger systems.

In the PDE algorithm used, for each element in the data array, two writes may be buffered at each time step before a fence operation is required. Figure 4 shows the relative execution time versus the number of elements per processor. The number of processors was kept constant. When there are multiple elements being executed upon by a thread, writes for all the elements may be buffered at each time step before a fence operation. For example when we have one element per processor, two writes may be buffered before a fence operation. If the number of elements per processor is increased to four, eight writes may be buffered before a fence operation and so on. Thus an increase in the number of elements per processor implies an increase in the size of the data set as well as an increase in the number of writes that may be buffered.

The distributed directory protocol with write buffering had the best performance for the experiments that were performed. In the case of small data sets, the performance of the centralized directory protocol was exceptionally poor due to contention at the centralized directory. The effect of this contention was reduced in the distributed directory protocol due to the use of the resources in a more distributed manner as explained in section 3.2. Write buffering improved the performance of both the protocols.

Figure 5 shows the effect of varying the SRAM to DRAM ratio. Write buffering was used for all the curves in figure 5. In the case of the centralized directory protocol, slower DRAM memory causes read and write misses to be serviced more slowly since all replies have to be sent by the slower main memory. The slower DRAM memory also causes the contention at the main memory to increase in the case of the centralized directory protocol, resulting in further degradation in performance. In the distributed directory protocol cache-to-cache transfers are used most of the time, and there is no centralized bottleneck at the main memory. Therefore, slower DRAM memory does not cause any significant degradation of performance. The distributed directory protocol thus allows the use of cheaper, larger and denser main memory without any significant degradation in performance.

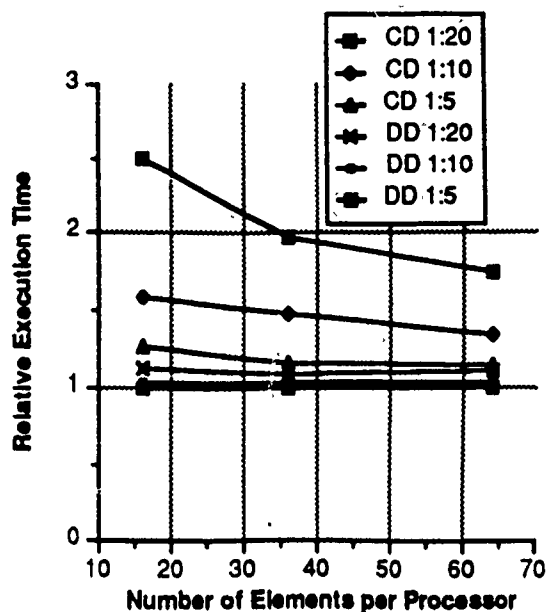


Figure 5: Effect of varying SRAM to DRAM ratio

4 Conclusions and Future Work

We have presented a new protocol for providing cache coherence in large scale shared memory machines. Besides savings in cost, the protocol also provides good performance. The protocol provides an efficient implementation of locks at minimal cost [11]. The scalability of the protocol provides us with enough reasons to conclude that the distributed directory protocol may be considered to be a viable solution for the cache coherence problem in large scale shared memory multiprocessors.

The distributed directory protocol presented in this paper is an invalidate based protocol but update based and hybrid protocols are possible to implement also. Hybrid protocols could update a part of the list and invalidate the rest. It would be useful to determine the tradeoffs of such variations.

Acknowledgements: We are thankful to Mike Flynn, Greg Byrd and Max Hailperin for their comments and suggestion and to the members of the DEC High Performance Systems Group and the Stanford Knowledge Systems Lab for their support.

References

- [1] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An evaluation of directory

- schemes for cache coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 281-289, 1988.
- [2] James Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):274-298, November 1986.
 - [3] Sandra Johnson Baylor and Faye A. Briggs. The effects of cache coherence on the performance of parallel PDE algorithms in multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages 233-236, 1989. Vol-I.
 - [4] Greg Byrd. Personal communication regarding talk given by Tom Knight at a workshop on ultra large scale message passing computers, 1987.
 - [5] Gregory Byrd, Nakul Saraiya, and Bruce Delagi. Multicast communication in multiprocessor systems. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages 196-200, 1989. Vol-I.
 - [6] Lucien M. Censier and Paul Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, c-27(12):1112-1118, December 1978.
 - [7] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 434-442, 1986.
 - [8] G. Pfister, W.Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, A. Norton, and J. Weiss. The IBM Research Parallel Processor prototype (RP3): Introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764-771, 1985.
 - [9] Christoph Scheurich and Michel Dubois. Correct memory operation of cache-based multiprocessors. In *Proceedings of the 14th International Symposium on Computer Architecture*, pages 234-243, 1987.
 - [10] Manu Thapar and Bruce Delagi. Design and implementation of a distributed directory cache coherence protocol. Technical report KSL-89-72, Stanford University, 1989.
 - [11] Manu Thapar and Bruce Delagi. Distributed cache coherence for large scale shared memory multiprocessor systems. Technical Report KSL-89-83, Stanford University, December 1989.
 - [12] Wolf-Dietrich Weber and Anoop Gupta. Analysis of cache invalidation patterns in multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243-256, 1989.

**MISSION
OF
ROME LABORATORY**

Rome Laboratory plans and executes an interdisciplinary program in research, development, test, and technology transition in support of Air Force Command, Control, Communications and Intelligence (C³I) activities for all Air Force platforms. It also executes selected acquisition programs in several areas of expertise. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C³I systems. In addition, Rome Laboratory's technology supports other AFSC Product Divisions, the Air Force user community, and other DOD and non-DOD agencies. Rome Laboratory maintains technical competence and research programs in areas including, but not limited to, communications, command and control, battle management, intelligence information processing, computational sciences and software producibility, wide area surveillance/sensors, signal processing, solid state sciences, photonics, electromagnetic technology, superconductivity, and electronic reliability/maintainability and testability.